
CSE6234

Software Proposal & Design Pattern Documentation

for

Grocery E-Commerce System

Version <1.0>

Tutorial Section: TT2L

Group Name: SuriaKLCC

Name	Student ID
Chew Xie Yang	1221304859
Anis Nur Hanani Binti Azhar	1221305140
Hafizul Faris bin Abdul Samad	1221304530
Fasya Eryna Binti Mokhtar	1201100273

Date: 15 Dec 2024

Content

Team Members.....	3
1 Introduction.....	4
1.1 Abstract.....	4
1.2 Problem Statement.....	4
1.3 Project Objectives.....	5
1.4 Literature Review.....	6
1.4.1 E-Commerce Challenges.....	6
1.5 Project and System Scope.....	8
2 System Overview.....	16
2.1 Generic Use Case.....	16
2.2 Class Diagram.....	17
2.3 ERD.....	18
2.4 Object Diagram.....	19
2.4.1 Customer Object Diagram.....	19
2.4.2 Retailer Object Diagram.....	22
3 Requirements.....	24
3.1 Software Design Concepts and Design Principles.....	24
3.1.1 Single Responsibility Principle.....	24
3.1.2 Open/Closed Principle.....	28
3.1.3 Liskov Substitution Principle (LSP).....	31
3.1.4 Interface Segregation Principle (ISP).....	32
3.1.5 Dependency Inversion Principle (DIP).....	33
4 Proposed Design Patterns.....	35
1.1 Factory Method (Customer Management).....	35
1.2 Composite (Shopping Cart Management).....	45
5 Conclusion and Suggestions.....	83
5.1 Conclusion.....	83
5.2 Suggestions.....	83
6 Bibliography/Reference.....	84
5.1 Bibliography.....	84
5.2 Reference.....	84

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Chew Xie Yang Anis Nur Hanani Hafizul Faris Fasya Eryna	First build	14/12/24

Team Members

Name
Anis Nur Hanani
Chew Xie Yang
Haifzul Faris
Fasya Eryna

1 Introduction

1.1 Abstract

The rapid development of technology has transformed different industries, including grocery shopping, by making online grocery shopping available to everyone through e-commerce. While the current e-commerce systems offer convenience and save time for their customers, there are certain drawbacks, such as the inability to provide personalized shopping experiences, which reduces customer satisfaction, disengages customers, and causes lost sales. Overcoming these challenges will help gain customer loyalty and thrive in a highly competitive market.

This project aims to develop an integrated e-commerce grocery platform that will enhance the customer experience. The system will include personalized features like product recommendations, user profiles to store their preferences, such as favorite products and dietary requirements, and customizable shopping templates for real-life scenarios. Other functionalities include real-time display of stock availability, integrated payment options (credit/debit cards, digital wallets, online banking), tracking orders and delivery updates, and an efficient notification system regarding order and stock status.

These features will be integrated into the proposed system to provide an easy, accessible, and pleasant shopping experience. This solution tries to solve the problem of current e-commerce systems to improve customer satisfaction and loyalty, setting a new benchmark in online grocery shopping.

1.2 Problem Statement

Technology has altered our way of life as it has developed over time. It has affected many departments/sectors, which includes the grocery shopping sector. Online grocery shopping has become widely accessible due to the quick expansion of e-commerce, offering the public a time-saving and convenient service.

However, despite the service it offers, e-commerce system implementation is fraught with challenges. These challenges include the absence of personalized shopping experiences, leading to lowered consumer happiness, causing them to feel disengaged and not "valued" or "understood," and resulting in lost sales chances. In this market that is becoming more and more competitive, by addressing this problem, it would resolve a huge issue in pre-existing e-commerce systems, and help enhance customer satisfaction as well as loyalty.

To address this issue, in our project, we aim to create a comprehensive e-commerce grocery that is tailored for each customer. The system will incorporate elements such as:

- Product recommendations
- User profiles to allow for a customer to record preferences (such as favorite products and dietary requirements),
- Feature numerous “templates” of grocery shopping for real-life scenarios to enhance accessibility of products.
-

By incorporating these elements, the system will enhance accessibility and customer satisfaction, which will help bring an experience that is user-friendly to the customers,

1.3 Project Objectives

As mentioned beforehand, the goal of this project is to be able to have a grocery e-commerce platform that can include personalized shopping experiences, and enhance customer satisfaction & loyalty.

By implementing this system, we will achieve:

- Enhance customer experience through personalization
- Design and display the stock information availability
- Provide customers with tracking and delivery status information
- Include payment options (ex. credit/debit cards, digital wallet, online banking).
- Facilitate efficient communication by implementing a notification system for updates on orders, and stock availability.

By having the objectives mentioned beforehand in mind, the software can be developed through this framework.

1.4 Literature Review

Introduction

The e-commerce industry has significantly transformed grocery shopping, providing time-saving and convenient solutions for customers. However, challenges persist in offering personalized shopping experiences and customer satisfaction. This literature review explores research that highlights key aspects of e-commerce challenges, customer personalization, and operational transparency which are vital for the development of our project platform.

1.4.1 E-Commerce Challenges

The study by **Santos et al. (2023)** discusses the dynamic nature of e-commerce and its critical issues. It highlights the importance of integrating advanced technologies like Artificial Intelligence (AI) for operational efficiency and blockchain for secure payment systems. Moreover, the study identifies challenges such as logistics optimization, customer data security, and seamless interface design. These insights align with our project's objective to enhance operational transparency and provide a strong system that provides customers' needs.

1.4.2 Customer Personalization

The paper by **Le et al. (2023)** emphasizes the critical role for personalized features in e-commerce platforms. The authors argue that personalized product recommendations and user interfaces enhance customer engagement and satisfaction. This study highlights that grocery e-commerce platforms must consider dietary preferences and real-life shopping scenarios which apply to our project's focus on user profiles and personalized grocery templates.

1.4.3 Operational Transparency

The study by **Cheng et al. (2020)** examines trust-building mechanisms which are fundamental to customer retention. The research emphasizes transparent stock availability, real-time delivery tracking, and secure payment gateways as main factors. Additionally, the integration of notification systems for updates on order status and product availability enhances customer trust. This aligns with our project's aim to include similar features for operational transparency and effective communication.

1.4.5 Conclusion

In conclusion, the studies provide valuable insights into improving e-commerce grocery platforms. They emphasize the importance of leveraging advanced technologies by offering personalized shopping experiences and building customer trust through operational transparency. These findings are essential to our project in supporting the goal to create a customer-centric and efficient system that provides satisfaction and loyalty.

1.5 Project and System Scope

Project Scope:

Requirements Gathering:

Objective: Understand business needs & user expectations.

Activities:

- Stakeholder interviews, surveys, and workshops for requirement gathering
- Identify primary users (e.g., customers, admins, delivery staff).
- Define requirements, both functional (search filters, payment integration, etc...) and non-functional (system scalability, response time, etc...)
- Create detailed requirements document or user stories for reference

Deliverables: Requirements specification document, user personas, use case diagrams.

System Design and Architecture:

Objective: Plan technical structure and workflows of platform.

Activities:

- Design overall system architecture (client-server model, microservices, etc).
- Create database schemas to manage product catalogs, orders, user data, etc.
- Define APIs for communication between frontend and backend.
- Develop UI/UX wireframes or prototypes to visualize interface.
- Address scalability, security, and performance considerations.

Deliverables: System architecture diagrams, wireframes, and design prototypes.

Development:

Objective: Build functional components of the platform.

Activities:

- Backend development: Implement server-side logic, database connections, and API endpoints.
- Frontend development: Develop responsive interfaces through use of frameworks such as React, Angular, or Vue.js.
- Implement third-party integrations such as payment gateways, delivery services, or analytics tools.
- Conduct unit testing for each component.

Deliverables: Functional backend and frontend systems, integrated features, and initial test results.

Testing:

Objective: Ensure system meets quality standards and works as intended.

Activities:

- **Functional Testing:** Verify that all features work according to requirements.
- **Usability Testing:** Assess user-friendliness and intuitiveness.
- **Performance Testing:** Test system's response time and scalability under various loads.
- **Security Testing:** Identify vulnerabilities, especially in payment processing and user authentication.
- **Regression Testing:** Ensure new updates don't break existing features.

Deliverables: Test cases, bug reports, and a quality assurance (QA) report.

Deployment:

Objective: Make the platform available for use in a live environment.

Activities:

- Set up a hosting environment (e.g., cloud servers like AWS, Azure).
- Configure domain names and secure connections (SSL certificates).
- Migrate system to staging for final user acceptance testing (UAT).
- Launch system into production environment.

Deliverables: Deployed platform, staging environment for UAT, and production-ready system.

Documentation:

Objective: Provide resources for system maintenance and user training.

Activities:

- Prepare technical documentation covering the architecture, APIs, and database.
- Write user manuals for platform administrators and customers.
- Provide troubleshooting guides for common issues.
- Maintain version control documentation for all code and updates.

Deliverables: Technical documents, user manuals, and a troubleshooting knowledge base.

Training and Support:

Objective: Equip users with knowledge and tools to manage the platform.

Activities:

- Conduct training sessions for admins on inventory and order management.
- Provide tutorials or walkthroughs for new users.
- Offer initial support for resolving user queries and system bugs.

Deliverables: Training schedules, recorded sessions or tutorials, and support contacts.

Post-Launch Maintenance:

Objective: Ensure long-term usability and reliability of the platform.

Activities:

- Monitor system performance and resolve any arising issues.
- Roll out updates to fix bugs and improve features.
- Scale server resources as user demand grows.
- Collect user feedback to guide future updates or features.

Deliverables: Maintenance logs, updated system versions, and user feedback reports.

System Scope:**User Roles and Accounts:**

Objective: Manage access and privileges for different users.

Features:

- **Customer accounts:**
 - Registration via email, phone, or social media.
 - Login, password reset, and profile management (e.g., addresses, preferences).
 - View order history and saved items.
- **Admin/Manager accounts:**
 - Role-based access control (e.g., admin vs. store manager).
 - Manage inventory, orders, and promotions.
 - Generate reports and access analytics.
- **Delivery personnel accounts (if applicable):**
 - Assign and manage delivery orders.
 - Track delivery status and customer feedback.

Deliverables: Role-based dashboards, secure authentication, and user account management.

Product Catalog Management:

Objective: Display and organize available products efficiently.

Features:

- Product categorization: Organize products into categories like "Fruits," "Vegetables," "Dairy," etc.
- Detailed product pages: Include name, description, price, discounts, availability, images, and ratings.
- Admin capabilities: Add, edit, or delete products and update inventory.
- Bulk upload of products using CSV or Excel files.

Deliverables: Dynamic product catalog, admin interface for catalog management.

Search and Filtering:

Objective: Allows quick-search for desired products.

Features:

- Search functionality: Full-text search for product names, brands, or categories.
- Filters: Sort products by price, rating, availability, and discounts.
- Auto-suggestions and spelling corrections for search queries.
- Faceted navigation: Combine multiple filters for precise results.

Deliverables: Advanced search and filtering capabilities integrated with the product catalog.

Shopping Cart and Checkout:

Objective: Facilitate a seamless purchasing experience.

Features:

- Shopping cart:
 - Add/remove/edit items in the cart.
 - Save cart for later (optional).
- Checkout process:
 - Address selection and delivery preferences.
 - Multiple payment methods (credit/debit cards, e-wallets, COD).
 - Apply coupons, promo codes, or loyalty points.
 - Order summary and confirmation screen.

Deliverables: Fully functional shopping cart and checkout system.

Order Management:

Objective: Manage the entire lifecycle of an order.

Features:

- Customers:
 - View current and past orders with statuses (e.g., "Processing," "Shipped," "Delivered").
 - Cancel or modify orders (within specified conditions).
- Admin:
 - Track orders and manage status updates.
 - Assign orders to delivery personnel.
- Notifications:
 - Email/SMS notifications for order confirmation, shipping, and delivery.

Deliverables: Order tracking system with real-time updates for both users and admins.

Inventory Management:

Objective: Monitor and manage stock levels.

Features:

- Real-time updates: Automatically update inventory levels after purchases.
- Notifications: Alerts for low stock or out-of-stock products.
- Admin panel:
 - Add/remove stock.
 - View detailed inventory reports.

Deliverables: Inventory management system integrated with product and order modules.

Delivery Management:

Objective: Ensure efficient delivery of orders to customers.

Features:

- Delivery scheduling: Allow customers to select delivery time slots.
- Delivery tracking: Real-time status updates for customers and admins.
- Integration with third-party delivery services or in-house logistics.
- Route optimization (if applicable) for delivery personnel.

Deliverables: Delivery tracking system with admin and customer views.

Payment Gateway Integration:

Objective: Facilitate secure and diverse payment options.

Features:

- Support for multiple payment methods:
 - Credit/debit cards.
 - Digital wallets (e.g., PayPal, Google Pay).
 - Cash on delivery (COD).
- Secure transactions: Use encryption (SSL/TLS) and PCI-DSS-compliant gateways.
- Refund and cancellation handling.

Deliverables: Integrated and secure payment processing system.

Analytics and Reporting:

Objective: Provide insights for data-driven decision-making.

Features:

- Sales trends: Analyze revenue and popular products over time.
- Customer behavior: Track user activity, preferences, and retention metrics.
- Inventory reports: Identify fast-moving and slow-moving items.
- Customizable reports: Allow admins to generate and download reports.

Deliverables: Analytics dashboard with detailed, customizable reports.

Customer Support:

Objective: Address customer queries and issues effectively.

Features:

- Chatbot: Provide automated answers to common questions.
- Live chat: Enable real-time interaction with support agents.
- Helpdesk system: Allow users to submit tickets for issues.
- FAQs: Provide a self-service section for frequently asked questions.

Deliverables: Integrated customer support tools and ticketing system.

Mobile Responsiveness:

Objective: Ensure the platform works seamlessly on all devices.

Features:

- Responsive design: Adapt UI/UX for desktops, tablets, and smartphones.
- Mobile-first design: Optimize for smaller screens without compromising usability.
- Optional: Dedicated mobile apps for iOS and Android.
- **Deliverables:** Fully responsive platform or mobile apps.

Outside Scope:**1. In-Store Management:**

- Physical store operations: Management of physical retail locations, such as POS systems, in-store customer management, or in-store inventory systems not connected to the e-commerce platform.
- Offline promotions: Local or in-store events and discounts unrelated to the online platform.

Reason for Exclusion: Under jurisdiction of physical store's management

2. Supplier Management Beyond Inventory:

- Vendor or supplier management: Negotiating contracts, onboarding suppliers, and managing supplier relationships outside the inventory updates for the platform.
- Procurement processes: Detailed management of purchase orders, delivery schedules from suppliers, and quality control at supplier sites.

Reason for Exclusion: Under jurisdiction of physical store's management

3. Manufacturing and Production Processes:

- Product sourcing or production: If the grocery items are sourced from manufacturers or farms, the details of procurement or manufacturing processes may be out of scope.
- Warehouse operations unrelated to e-commerce: Internal logistics and processes such as receiving bulk stock and its physical placement within warehouses unless integrated with inventory management.

Reason for Exclusion: Under jurisdiction of suppliers.

4. External Marketing and Branding:

- Offline marketing campaigns: Flyers, billboards, or in-store advertising.
- Social media management: Tools or strategies for social media marketing not directly tied to the platform.
- Brand design: Creating brand identity elements such as logos, taglines, or print designs, unless they are part of UI/UX design considerations.

Reason for Exclusion: Under jurisdiction of retailer's management

5. Advanced Customer Personalization:

- Hyper-personalization features: Such as tailored recommendations based on real-time behavior or dynamic pricing systems that may go beyond initial requirements.

Reason for Exclusion: Possible upgrade feature

- Extensive loyalty program integrations: Beyond simple coupon codes or basic point accumulation.

Reason for Exclusion: Under jurisdiction of retailer's business management

6. Marketplace Features:

- Multi-vendor marketplace functionalities: Allowing third-party sellers to register, list, and manage products independently, unless explicitly required.

Reason for Exclusion: Does not fit the software concept

7. Non-core Administrative Functions:

- Human resource management: Employee payroll, hiring, or performance tracking for admins or delivery personnel.
- Legal and compliance management: Detailed tracking of tax compliance, legal contracts, or corporate audits.

Reason for Exclusion: Under jurisdiction of the retailer's business management

8. Internationalization:

Multi-language or multi-currency support: If the platform is not intended for international users.

Cross-border delivery logistics: Complex delivery features for cross-country operations.

Reason for Exclusion: Legal complexity and business specific

9. Advanced Customer Engagement Tools:

- Gamification elements: Like reward systems for engagement or challenges.
- Community forums or reviews unrelated to product ratings.

Reason for Exclusion:

10. Specialized Services:

- Subscription models: Recurring delivery or subscription box services unless explicitly required.

Reason for Exclusion: Possible future upgrade.

- Dynamic price negotiations: For bulk buying or business-to-business orders.

Reason for Exclusion: Beyond the general concept of the software.

2 System Overview

2.1 Generic Use Case

The grocery e-commerce system will provide a platform for customer account management, product catalog management, shopping cart management, order management, payment processing, and delivery. It will also include features such as notification, and personalized product categories. It will be integrated to improve customer satisfaction and loyalty by making the grocery shopping experience seamless while providing real-time status for stock availability, order status, and delivery status.

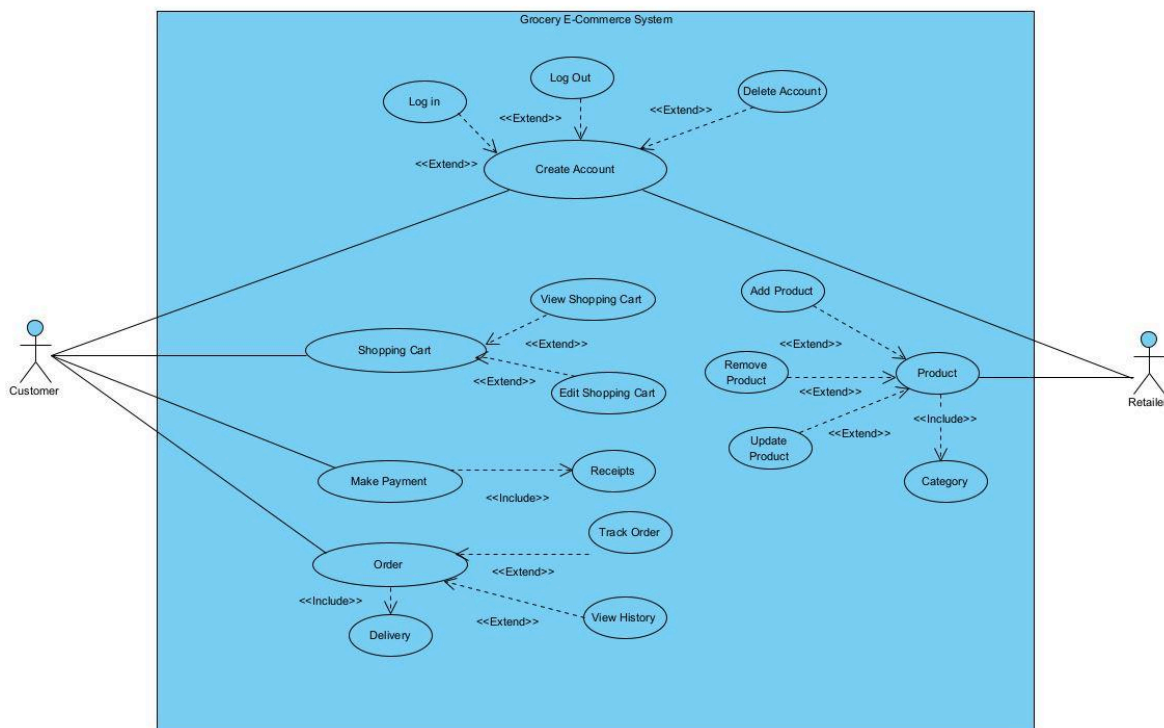


Figure 1.0: Use case Diagram

2.2 Class Diagram

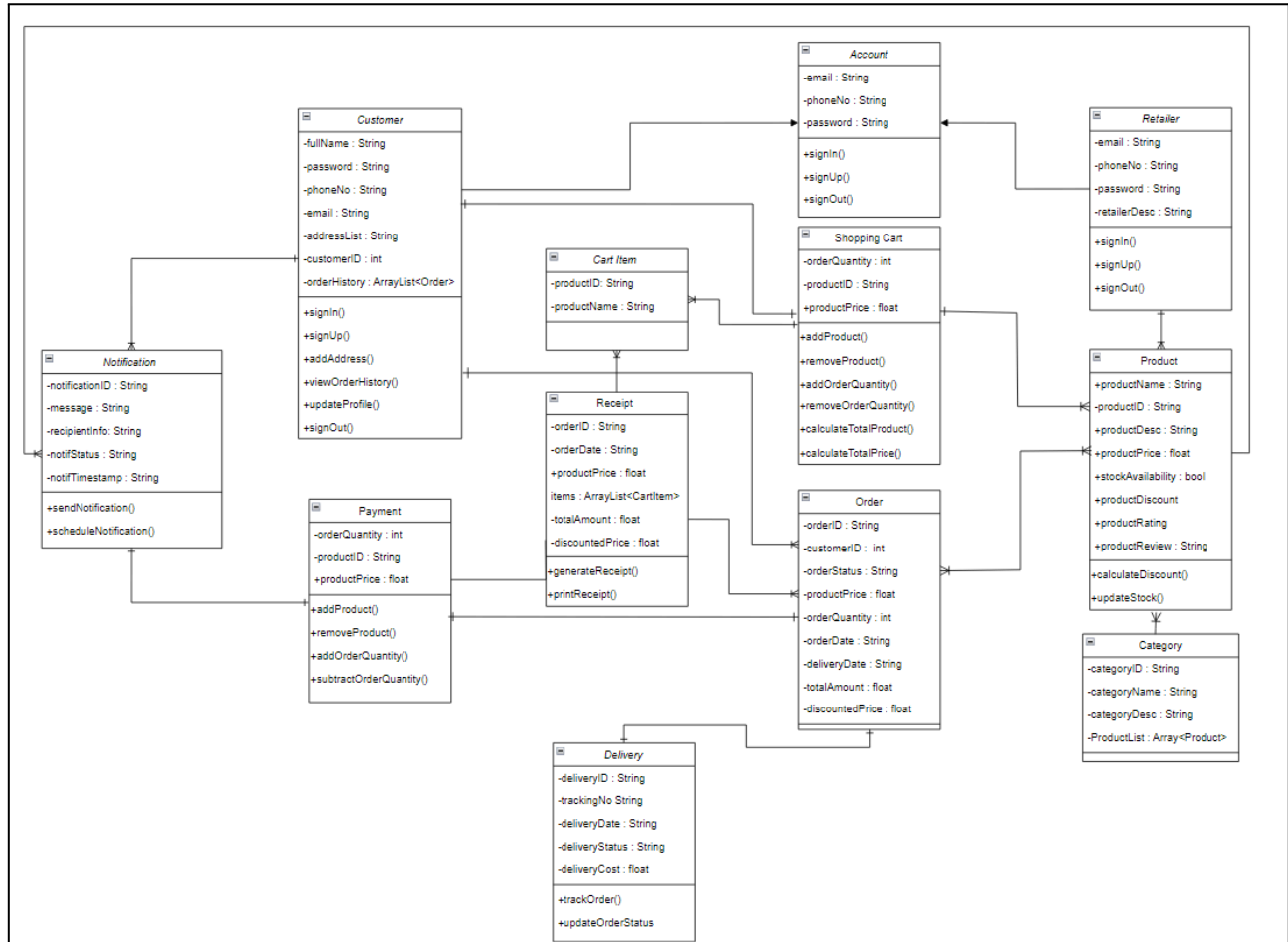


Figure 2.0: Class Diagram

2.3 ERD

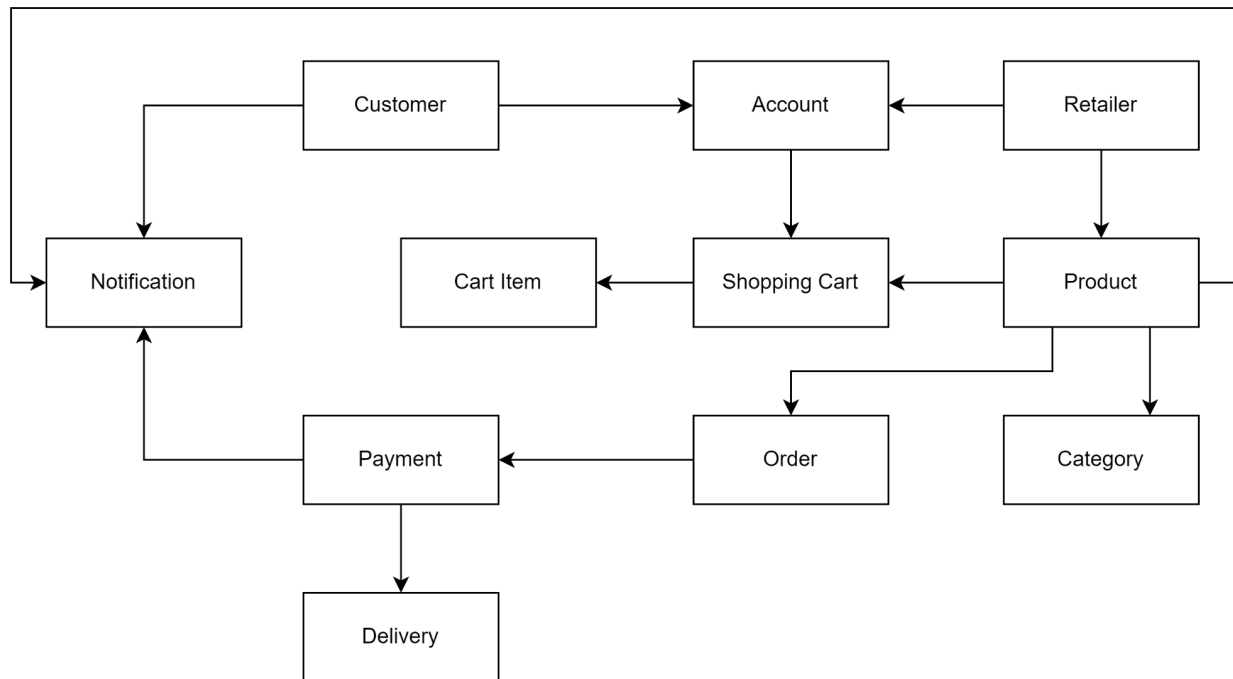


Figure 3.0: Entity Relationship Diagram (ERD)

2.4 Object Diagram

2.4.1 Customer Object Diagram

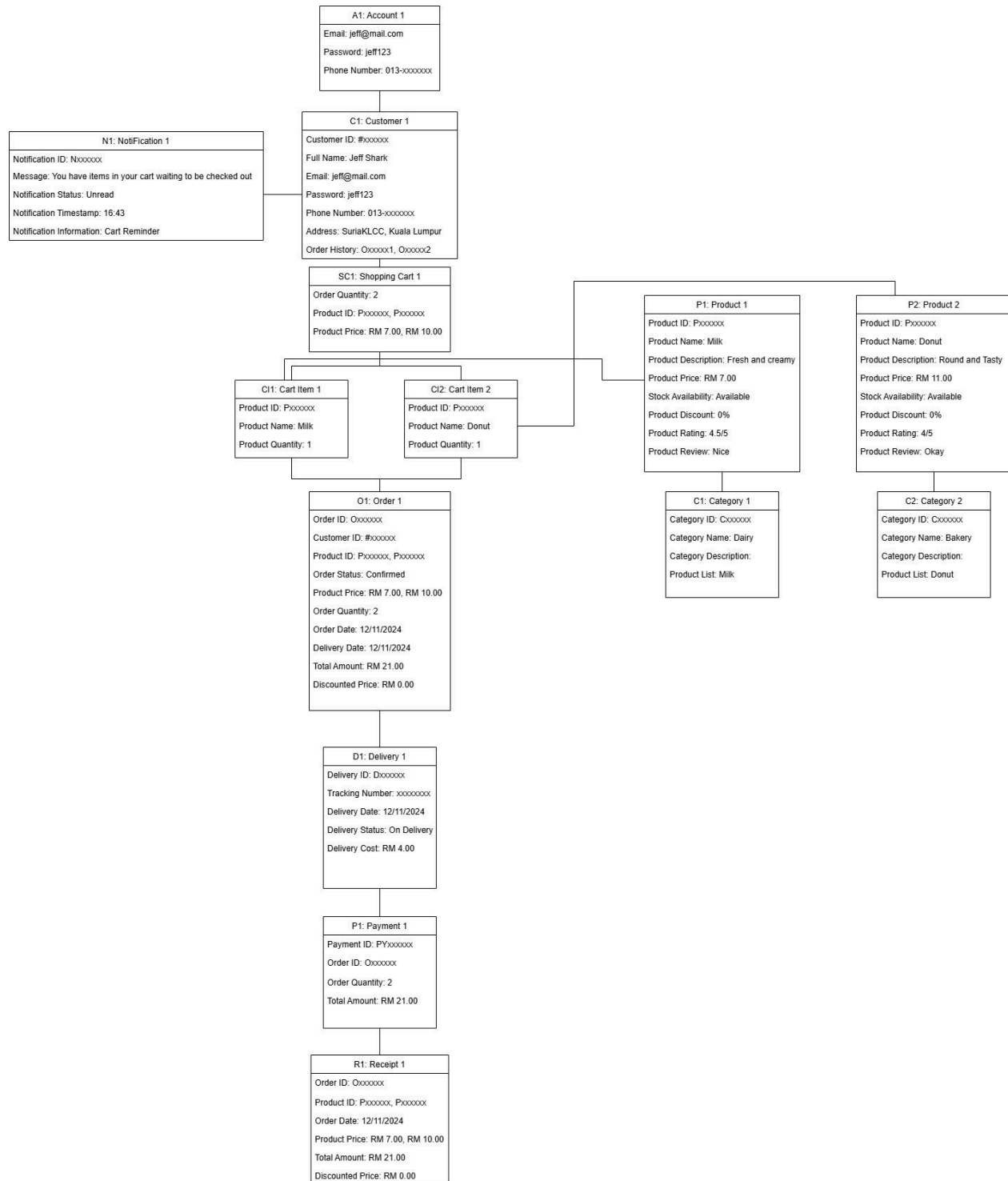


Figure 4.0: Customer Object Diagram

Account

The Account object stores the essential credentials and personal information of the user for them to access the system. Each customer is uniquely differentiated through their account. In Figure 4.0, Account (A1) is connected to Customer (C1)

Customer

The Customer object contains the customer's personal details and preferences for an easier shopping experience, as well as, allowing the users to track their previous orders. In Figure 4.0, Customer (C1) is connected to Notification (N1), Shopping Cart (SC1)

Shopping Cart

The Shopping Cart object is a virtual container that holds items the customer intends to purchase and displays the prices of the items included in the cart. In Figure 4.0, Shopping Cart (SC1) is connected to Cart Item(CI1) and Cart Item(CI2)

Product

The product objects are the items that show availability for the customer if they would like to purchase. The products are connected to the category for easier viewing for the user once they filter what exact items they want. In Figure 4.0, Product (P1) is connected to Cart Item(CI1). While, Product (P2) is connected to Cart Item(CI2)

Category

Category objects aid the user in their shopping experience, making it easier to filter the specific items. In Figure 4.0, Category(CG1) is connected to Product (P1). Whereas, Category(CG2) is connected to Product (P2), giving the example that not every product shares the same category.

Cart Item

To present individual products contained in the Shopping Cart(SC1), the Cart Item (CI1) object is used to represent each individual product, this way, the Shopping Cart can accurately show the products included. Multiple Cart Item (CI1 & CI2) objects can exist within a single Shopping Cart. In Figure 4.0, Cart Item(CI1) and Cart Item(CI2) is connected to Shopping Cart (SC1) and Order (O1)

Notification

The Notification object's purpose is to send a message or a reminder to the customer. Such as, in the context of Figure 4.0, it would send a reminder to the user if there are untouched items in their shopping cart In Figure 4.0, Notification (N1) is connected to Customer (C1)

Order

The Order object represents a confirmed purchase made by the customer. It helps keeping the customer informed regarding their order status. In Figure 4.0, Order (O1) is connected to Cart Item(CI1, CI2) and Delivery (D1)

Delivery

The Delivery object helps in giving status on the whereabouts of the customer's order. In Figure 4.0, Delivery (D1) is connected to Order (O1) and Payment (P1)

Payment

The Payment object stores information regarding the transaction of order. This object makes sure that the financial aspects of every transaction are well documented and integrated into the system as a whole. In Figure 4.0, Payment (P1) is connected to Delivery (D1) and Receipt (RC1)

Receipt

The Receipt object provides a detailed breakdown of the transaction for the customer. The object serves as proof of purchase. In Figure 4.0, Receipt (RC1) is connected to Payment (P1)

2.4.2 Retailer Object Diagram

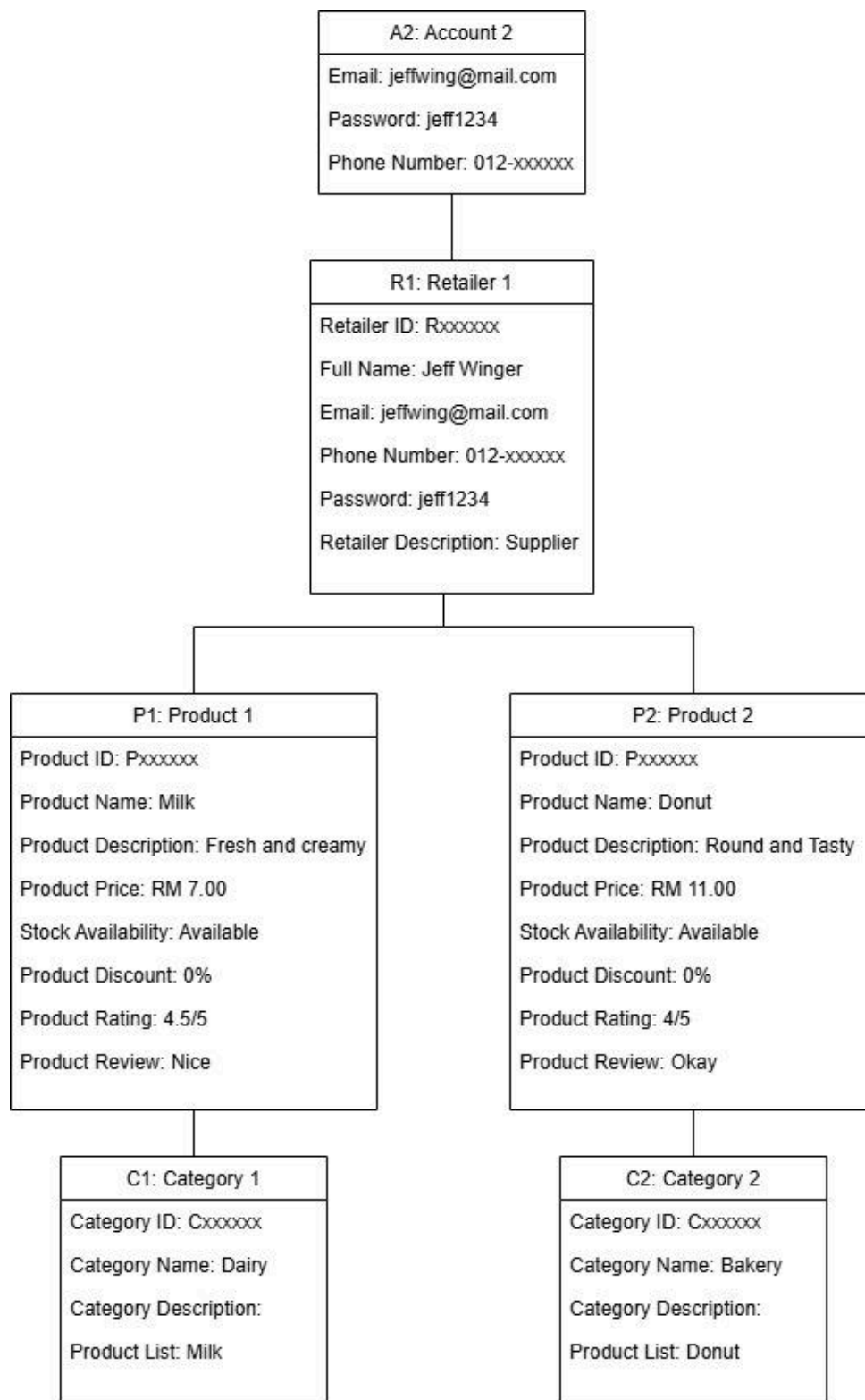


Figure 4.1: Retailer Object Diagram

Account

The Account object contains the essential credentials and personal information of the user for them to access the system. Retailers would have their own accounts to access the system. In Figure 4.1, Account (A1) is connected to Customer (C1)

Retailer

The Retailer object represents the entity responsible for supplying and managing products in the system. In Figure 4.1, Retailer (R1) is connected to Cart Item(CI1). While, Product (P2) is connected to Cart Item(CI2)

Product

The product objects are the items that the retailers are supplying for the customers to purchase. The products are connected to the category for easier viewing for the user once they filter what exact items they want. In Figure 4.1, Product (P1) is connected to Cart Item(CI1). While, Product (P2) is connected to Cart Item(CI2)

Category

In the context of retailer, Category object is for retailers to label the products for customer to filter items they are looking for. In Figure 4.1, Category(CG1) is connected to Product (P1). While, Category(CG2) is connected to Product (P2)

3 Requirements

3.1 Software Design Concepts and Design Principles

In order to maximize the efficiency of our system project, it should adhere to the SOLID principles, and include software design concepts such as “abstraction, modularity, encapsulation, functional independence, refinement, refactoring,” and “architecture.”

- 3.1.1 Single Responsibility Principle (SRP)
- 3.1.2 Open/Closed Principle
- 3.1.3 Liskov Substitution Principle (LSP)
- 3.1.4 Interface Segregation Principle (ISP)
- 3.1.5 Dependency Inversion Principle (DIP)

3.1.1 Single Responsibility Principle

Each class/method should only have one reason to change, or only one responsibility to be involved in. By establishing clear cut roles for each class/method, the overall code can be organized into manageable portions, allowing ease of alteration with minimal risk to affecting other functions. This improves the maintainability of the software by making modifications easier.

To do so, each class should focus primarily on only one core functionality, such as Payment, Shopping Cart, Orders, Product, and many others. The Payment class, for instance, would only be responsible for the function of payment and no other functions.

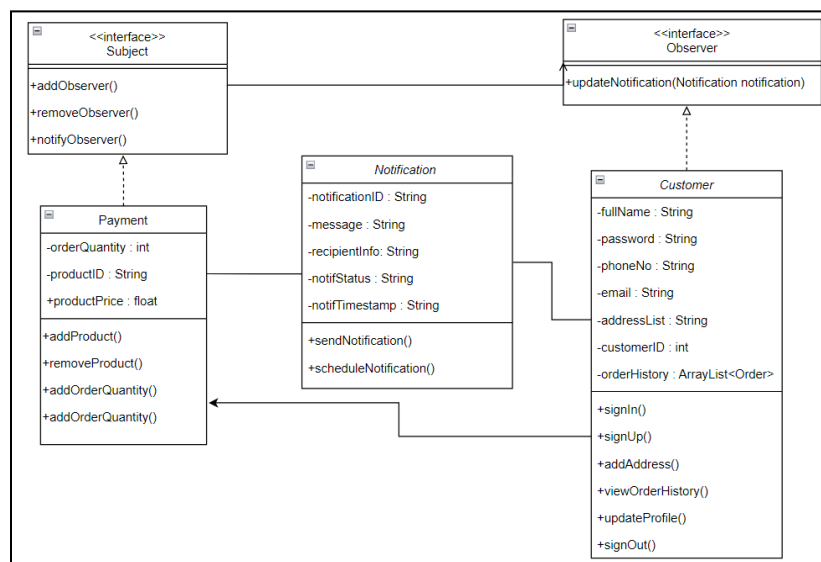


Figure 5.0.0: Sample Observer Class Diagram (Notifications)

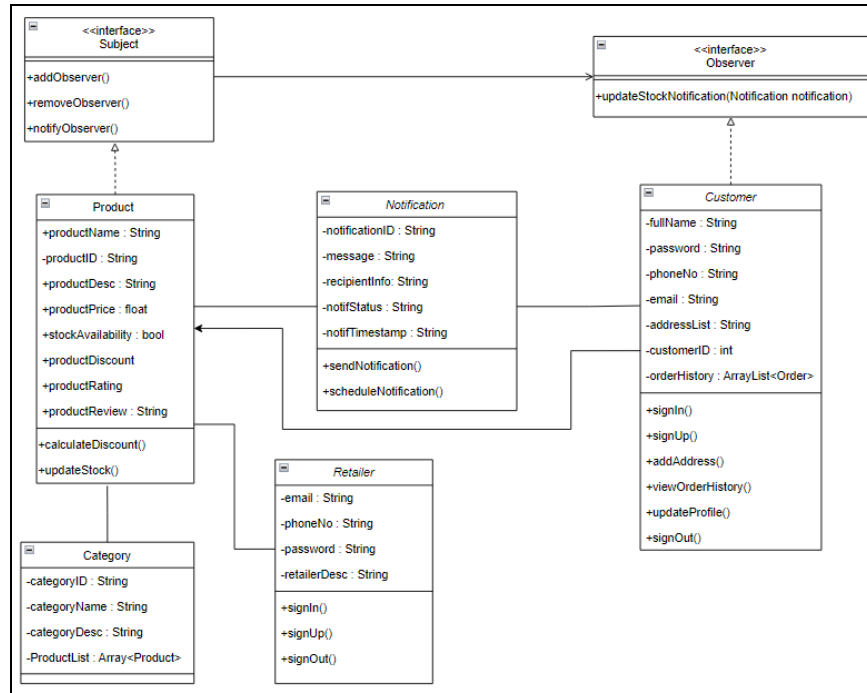


Figure 5.0.1: Sample Observer Class Diagram (Product Stock & Management)

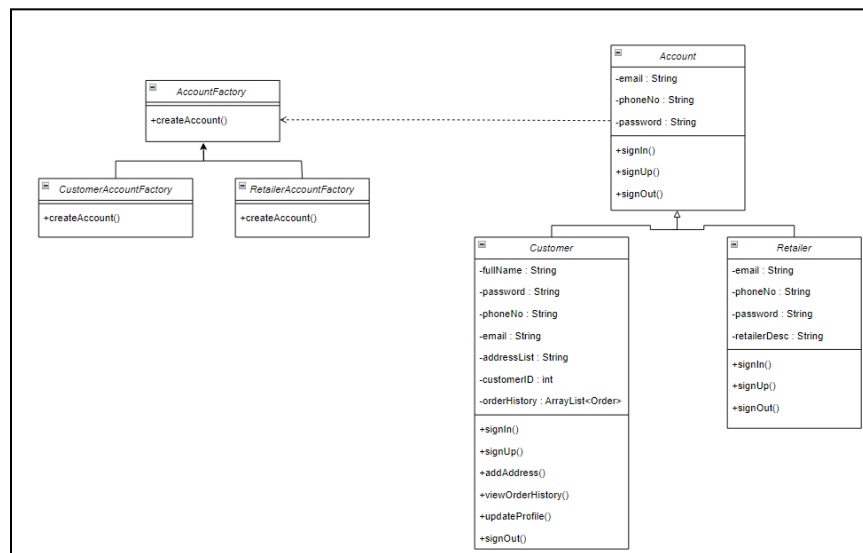


Figure 5.0.2: Sample Factory Class Diagram

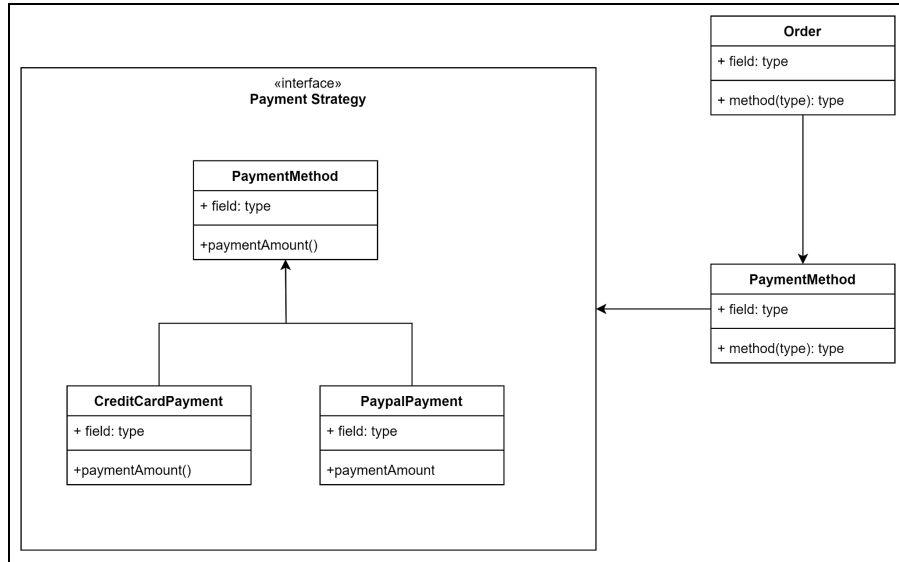


Figure 5.0.3: Sample Strategy Class Diagram

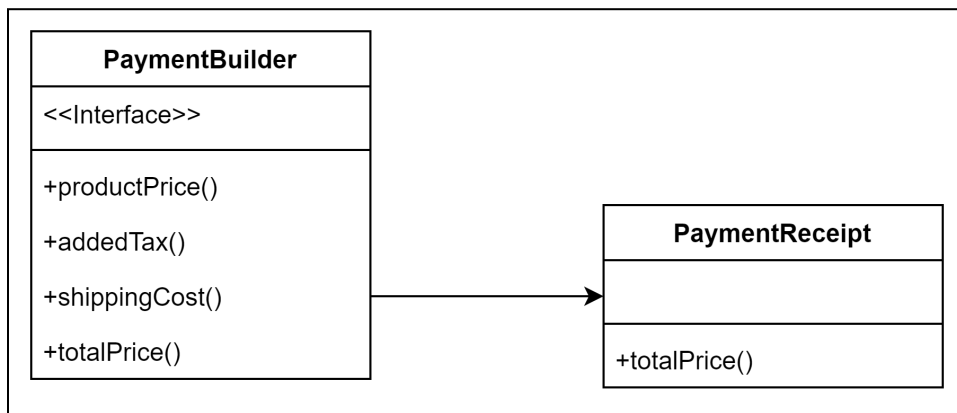


Figure 5.0.4: Sample Builder Class Diagram

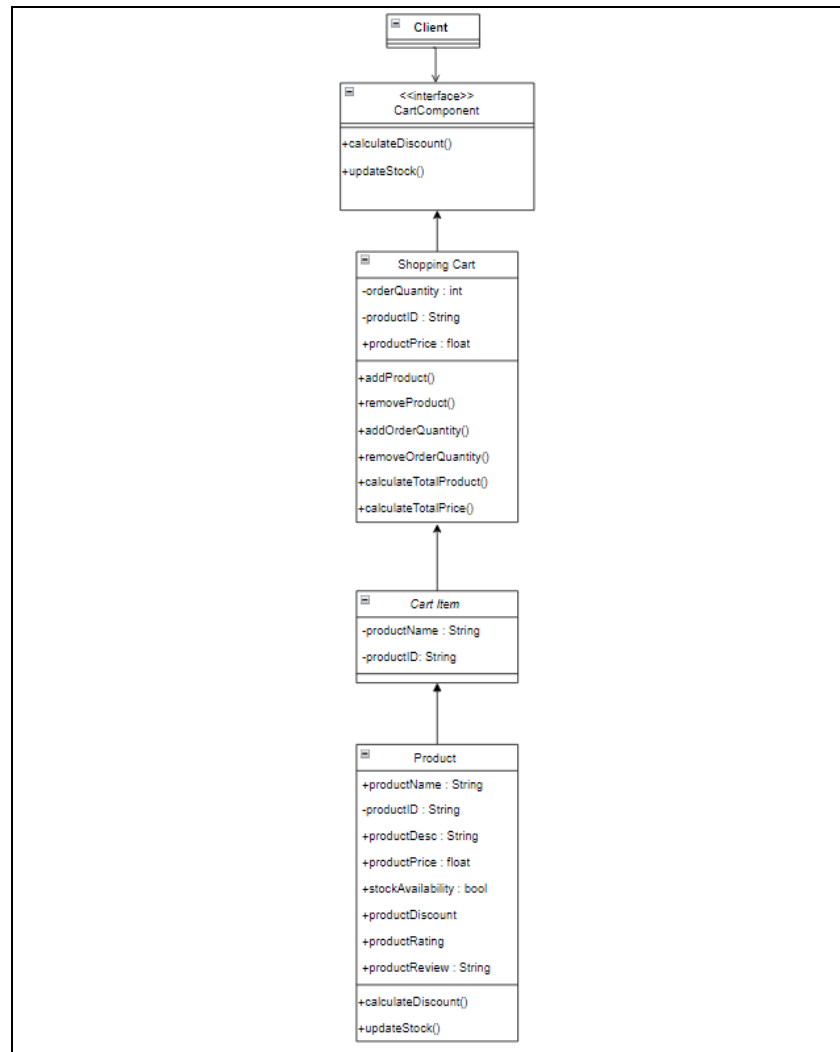


Figure 5.0.5: Sample Composite Class Diagram

3.1.2 Open/Closed Principle

Each class/module/function should only be open for extensions, but closed for modifications. They should be flexible enough to accommodate changes when business requirements change without breaking the existing code, and hence not requiring dependent codes to change, reducing testing time to allow focus on new code changes. To do so, we can either use inheritance or interface and abstract classes, to streamline the codes.

For instance, for a Payment Gateway interface, implementing classes for different payment methods that utilizes it, ensures that addition of new payment methods will only require extension of the system.

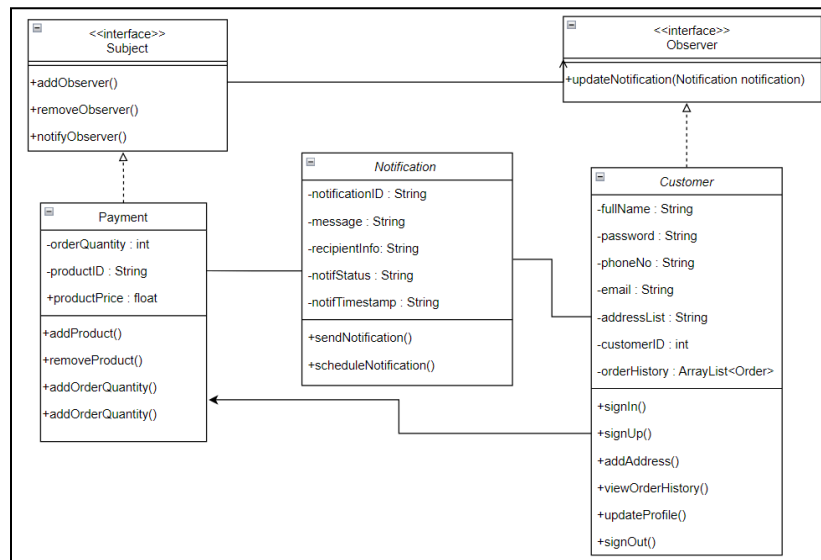


Figure 5.1.0: Sample Observer Class Diagram (Notifications)

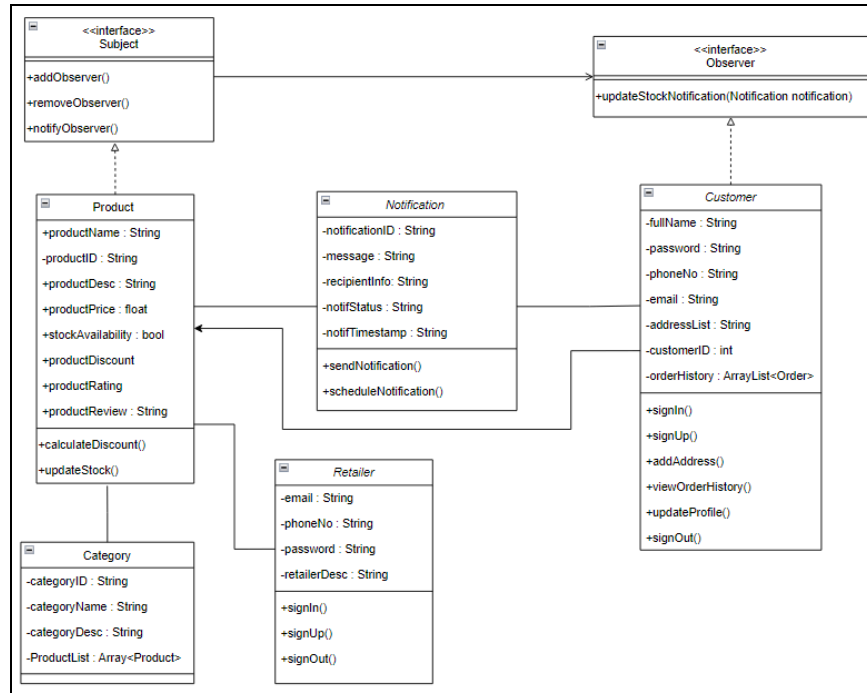


Figure 5.1.1: Sample Observer Class Diagram (Product Stock & Management)

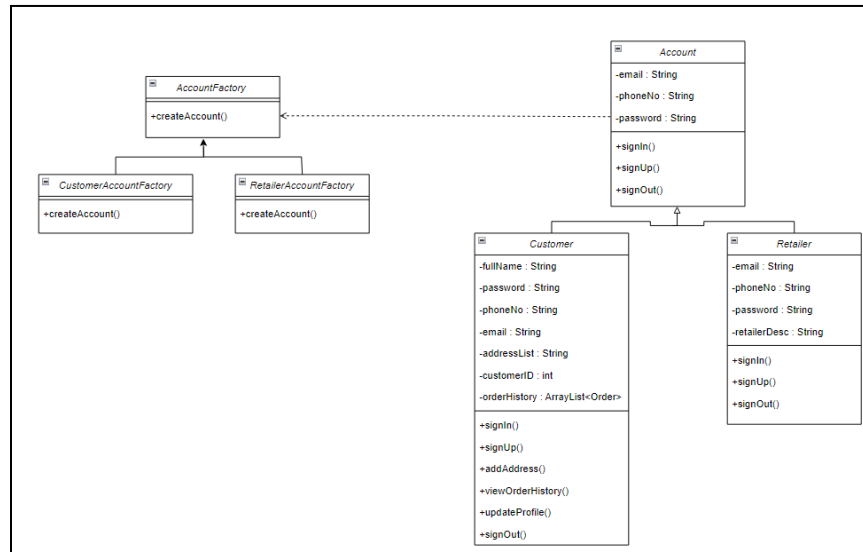


Figure 5.1.2: Sample Factory Class Diagram

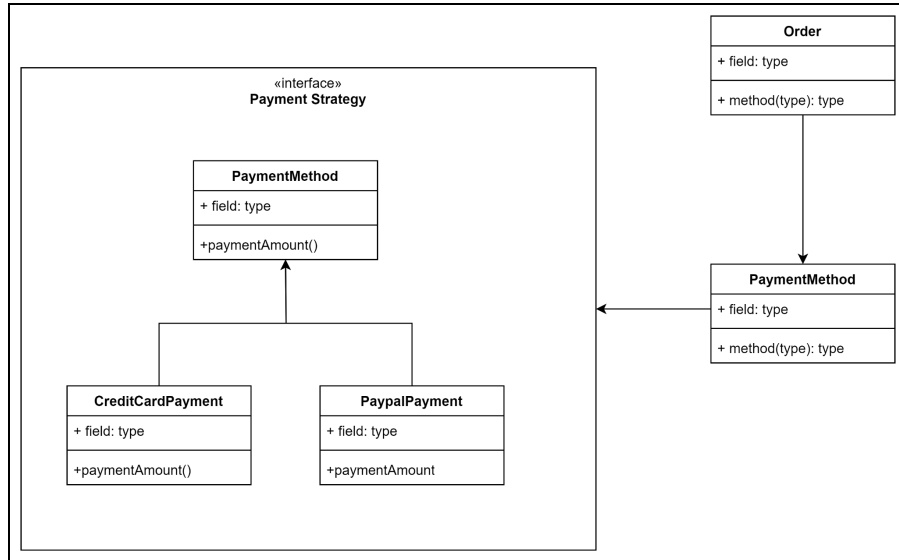


Figure 5.1.3: Sample Strategy Class Diagram

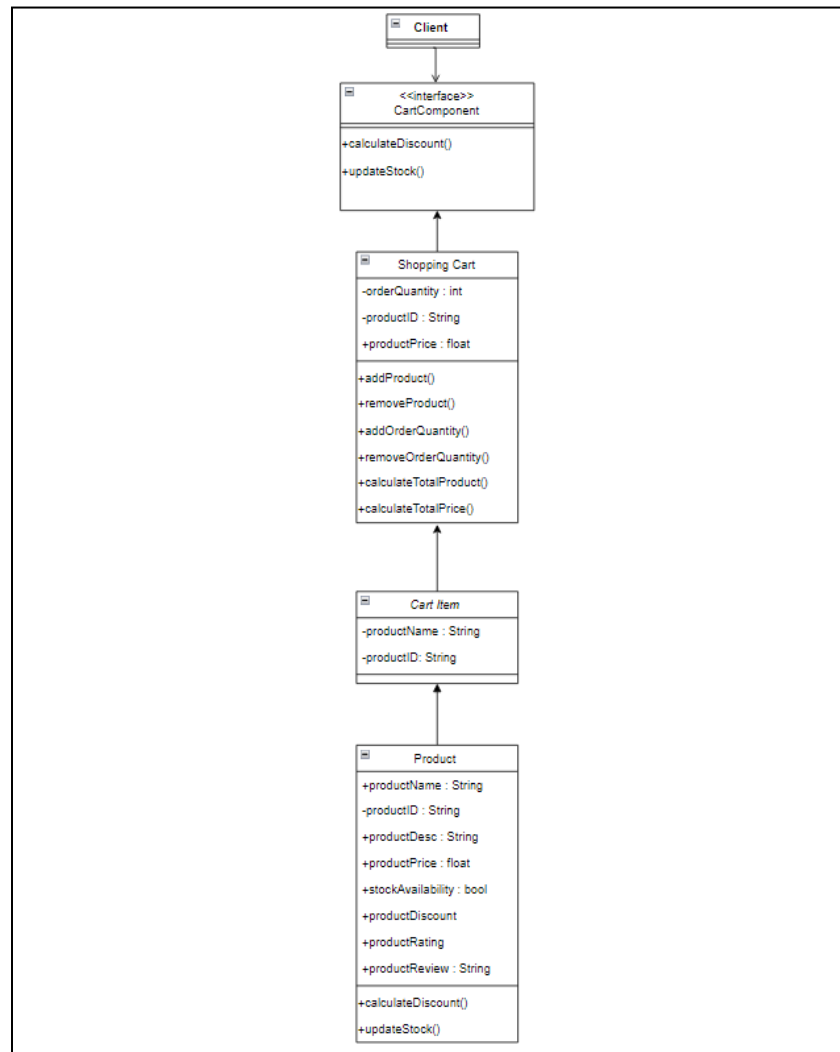


Figure 5.1.4: Sample Composite Class Diagram

3.1.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that any object of a superclass should be replaceable with objects of its subclasses without breaking the application itself. This requires that objects in the subclasses should behave in the same way as the objects in the superclass. This ensures consistent behavior among superclasses and subclasses.

For instance, with the singular class Product, subclasses such as Dairy, Grain, Fruit, Vegetable, Meat and Seafood, can be created, which will function similarity as the superclass Product.

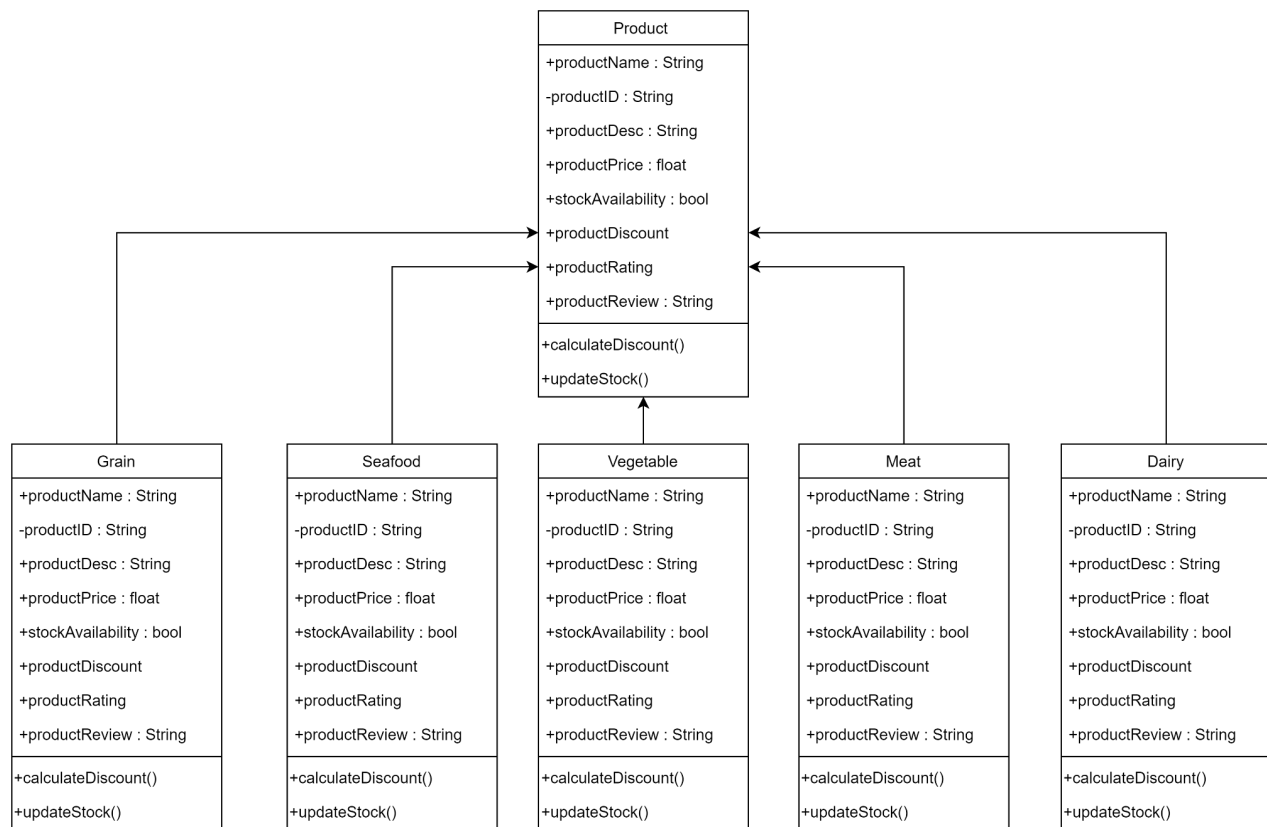


Figure 5.2: Sample Product Class Diagram

3.1.4 Interface Segregation Principle (ISP)

No user should have to use methods that they don't need to use. To ensure a streamlined interface that is user friendly, multiple client interfaces that are task-specific should be implemented. This not only follows the Single Responsibility principle, but also ensures satisfaction of client or customer in software use.

Separating the checkout interface into manageable components [payment, shopping cart, orders] allows the user to focus on one part of the ordering process at the time, allowing minimal intrusion to the customer experience.

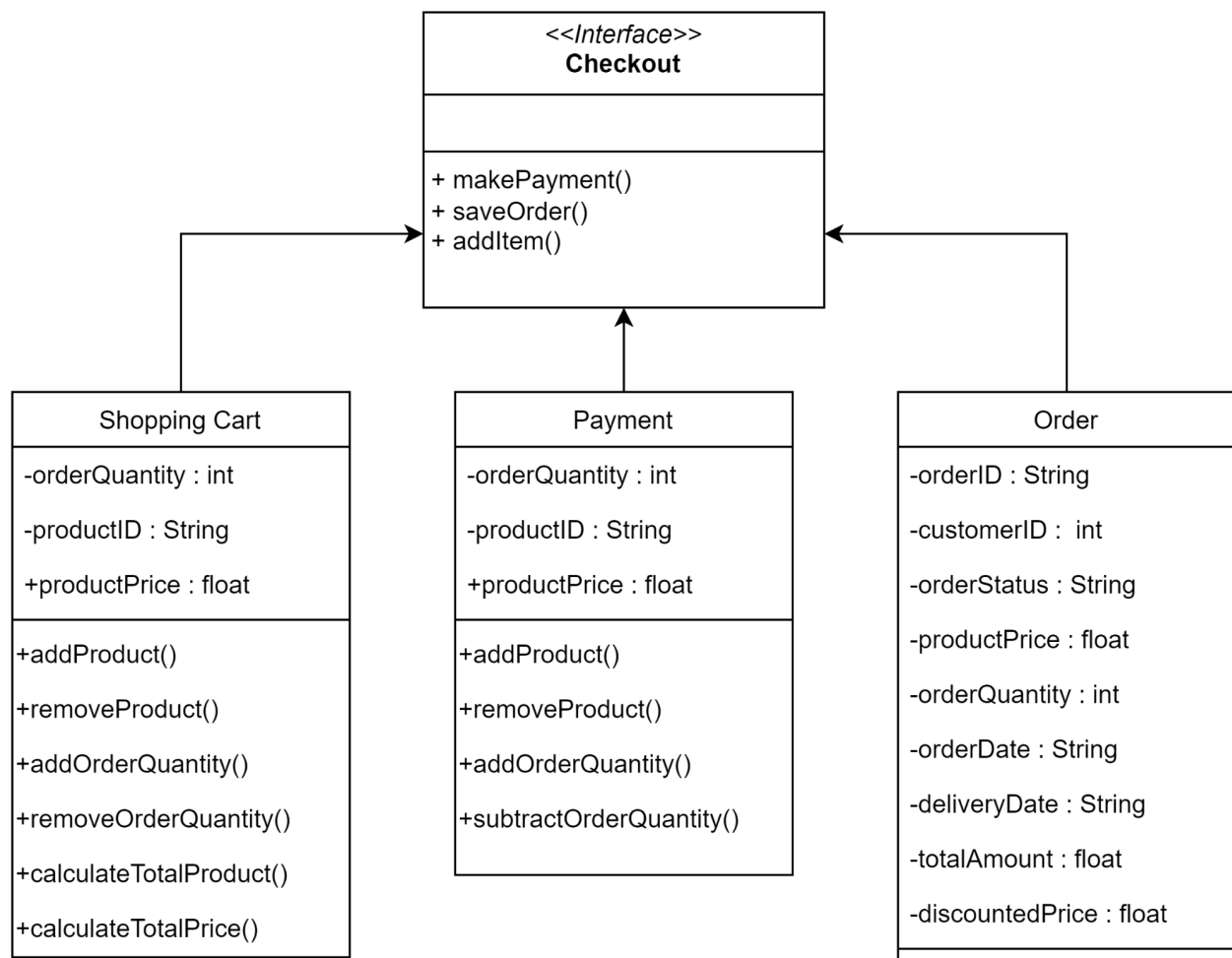


Figure 5.3: Sample Checkout Interface Diagram

3.1.5 Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules, while both should depend on abstractions. By focusing on abstraction dependency instead of concrete implementations, a more flexible and decoupled code can allow ease of implementation change without affecting other parts of the code. This ensures greater adaptability within the software development process.

By establishing an interface for delivery notifications, where notifications can be sent by email or text message, we can inject the concrete implementations to modules such as Delivery. This allows decoupling high-level business logic from specific implementations, allowing greater adaptability to changes in the software.

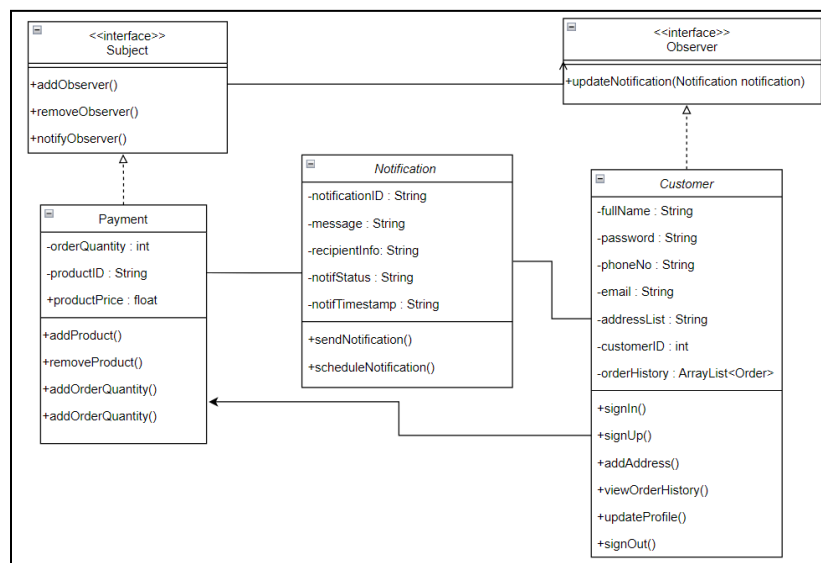


Figure 5.4.0: Sample Observer Class Diagram (Notifications)

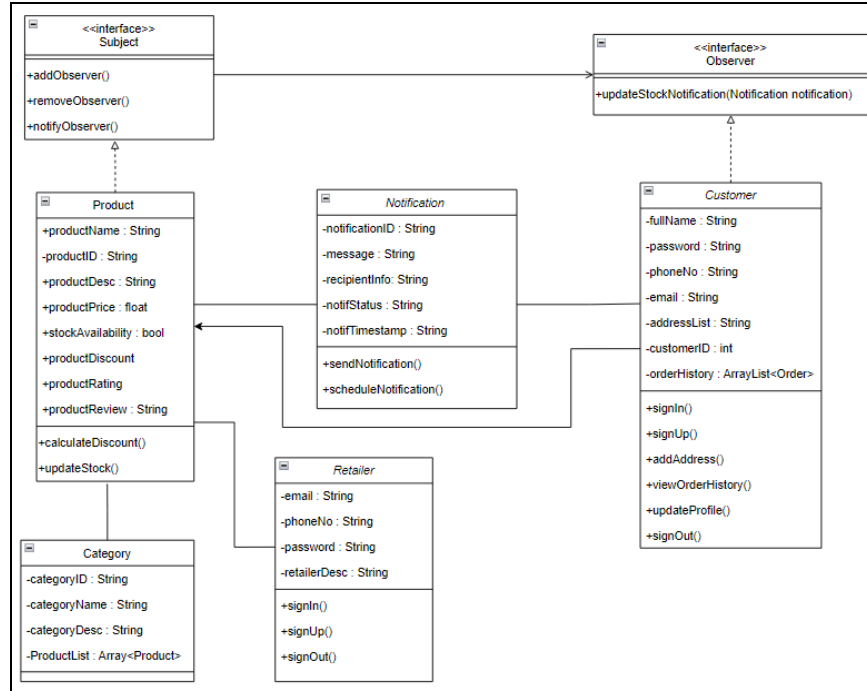


Figure 5.4.1: Sample Observer Class Diagram (Product Stock & Management)

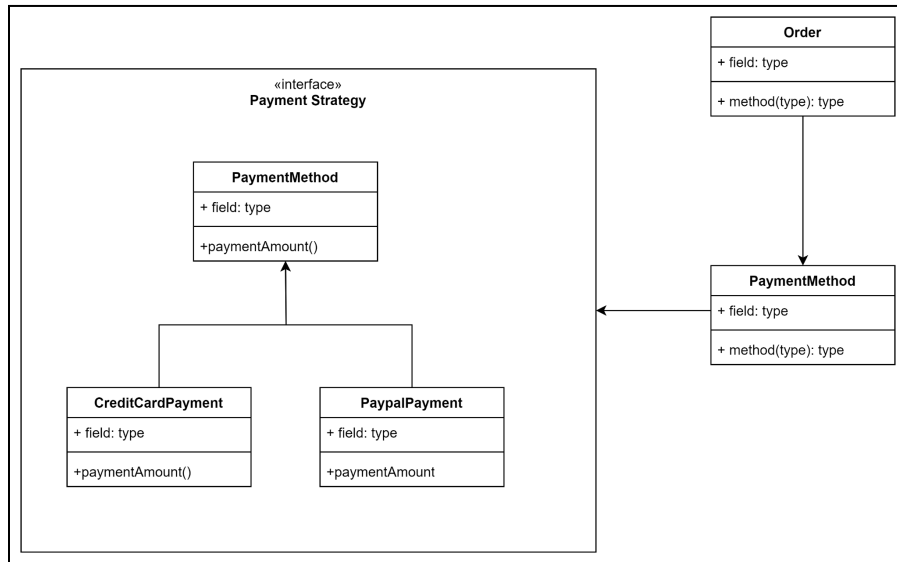


Figure 5.4.2: Sample Strategy Class Diagram

4 Proposed Design Patterns

1.1 Factory Method (Customer Management)

Function or Software Component Affected

The factory method can be used to create customer accounts based on different user types (e.g. regular, admin). In the future, if the company decides to expand on making various types of users, this would be a good choice as it would instantiate different types of customers.

Work/Data Flow

1. The client requests to create a customer account.
2. The parent class (that abstracts the instantiation logic of the account type) creates an account object.
3. Factory method returns the appropriate account type object.

Benefits

- Centralized control of object creation, making the code easier to maintain and extend.
- It helps simplify instantiation of different customer types or attributes.
- It can help provide flexibility when creating objects based on different inputs/conditions.

Limitations

- The Factory Method introduces additional classes to handle object creation, requiring more complexity (which is not suitable in smaller systems).
- The pattern may be a bit too much if customer types are simple (ex. RegularCustomer, AdminCustomer account) and don't require differentiation.

Sample Class Diagram

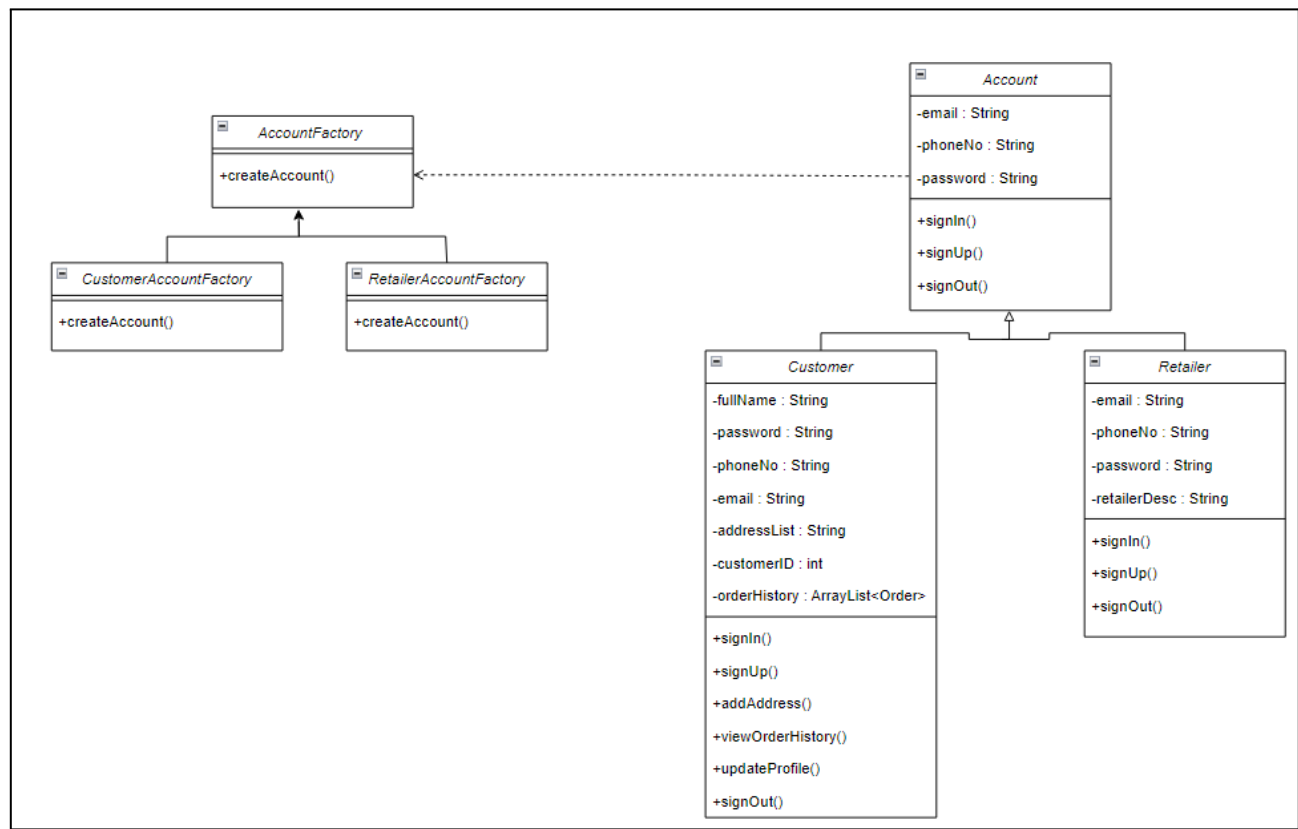


Figure 6.0: Sample Factory Class Diagram

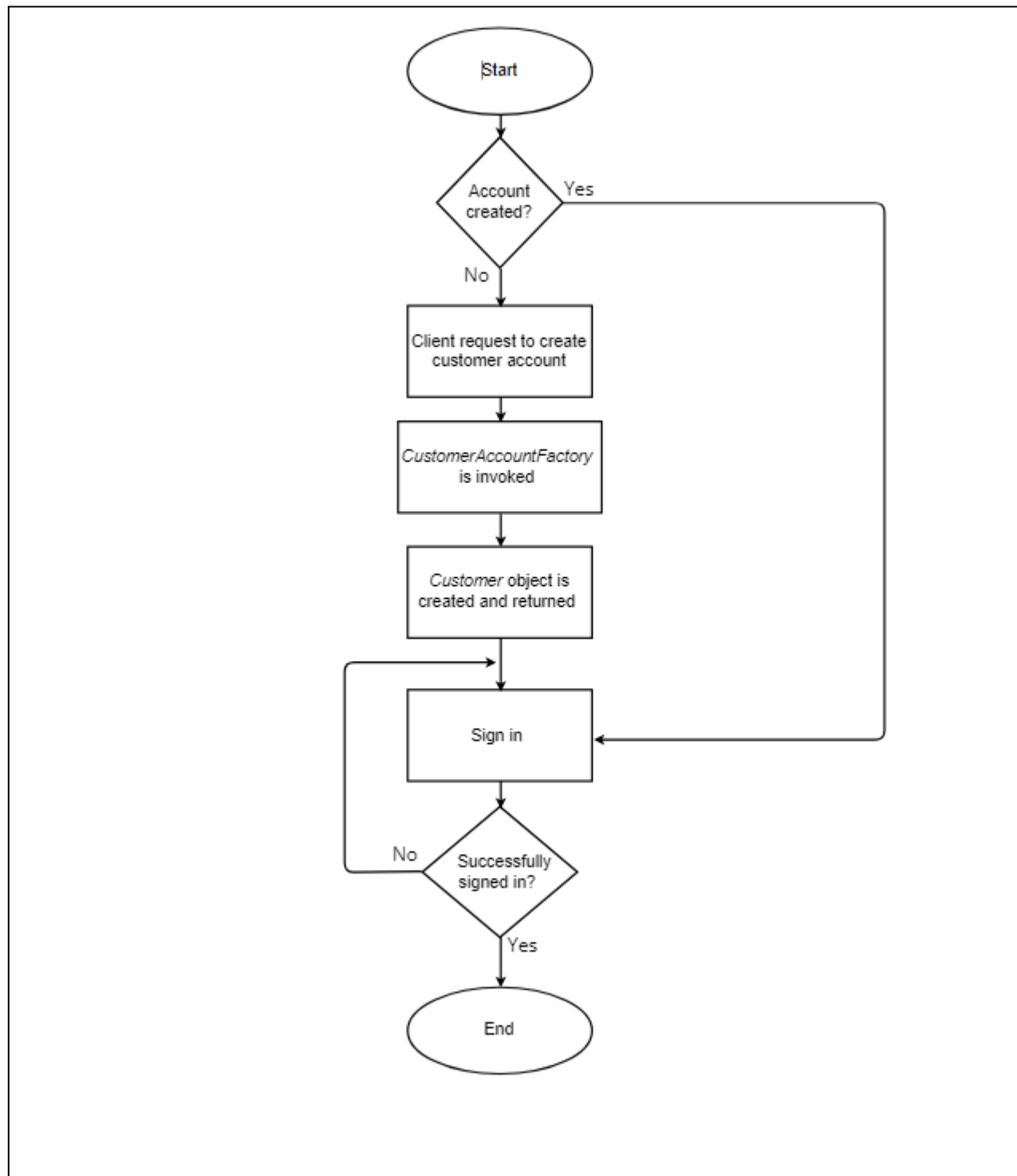
Sample Workflow Diagram

Figure 6.1: Sample Workflow Diagram for Factory

Sample Potential Code

```
from abc import ABC, abstractmethod
import csv
import tkinter as tk
from tkinter import messagebox

# Abstract Factory class
class AccountFactory(ABC):
    @abstractmethod
    def create_account(self):
        pass

# Concrete Factory for Customer Accounts
class CustomerAccountFactory(AccountFactory):
    def create_account(self):
        return Customer()

# Concrete Factory for Retailer Accounts
class RetailerAccountFactory(AccountFactory):
    def create_account(self):
        return Retailer()

# Abstract Account class
class Account(ABC):
    def __init__(self, email, phone_no, password):
        self.email = email
        self.phone_no = phone_no
        self.password = password

    def sign_in(self, email, password):
        return self.email == email and self.password == password

# Customer class inheriting from Account
class Customer(Account):
    def __init__(self, email="", phone_no="", password="", full_name="", customer_id=0):
        super().__init__(email, phone_no, password)
        self.full_name = full_name
```

```
        self.customer_id = customer_id

# Retailer class inheriting from Account
class Retailer(Account):
    def __init__(self, email="", phone_no="", password="",
retailer_desc=""):
        super().__init__(email, phone_no, password)
        self.retailer_desc = retailer_desc

# Save account to CSV
def save_account_to_csv(account, account_type):
    filename = f"{account_type}_accounts.csv"
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        if account_type == "customer":
            writer.writerow([account.email, account.phone_no,
account.password, account.full_name, account.customer_id])
        elif account_type == "retailer":
            writer.writerow([account.email, account.phone_no,
account.password, account.retailer_desc])

# Load accounts from CSV
def load_accounts_from_csv(account_type):
    filename = f"{account_type}_accounts.csv"
    accounts = []
    try:
        with open(filename, mode='r', newline='') as file:
            reader = csv.reader(file)
            for row in reader:
                if account_type == "customer":
                    accounts.append(Customer(email=row[0],
phone_no=row[1], password=row[2], full_name=row[3],
customer_id=int(row[4])))
                elif account_type == "retailer":
                    accounts.append(Retailer(email=row[0],
phone_no=row[1], password=row[2], retailer_desc=row[3]))
    except FileNotFoundError:
        print(f"No existing {account_type} accounts found. Starting
fresh.")
```

```
        return accounts

# Validate account
def validate_account(email, password, accounts):
    for account in accounts:
        if account.email == email and account.password == password:
            return True
    return False

# GUI Workflow Implementation
def main_workflow():
    def on_sign_in():
        account_type = account_type_var.get()
        email = email_entry.get()
        password = password_entry.get()

        if account_type not in ["customer", "retailer"]:
            messagebox.showerror("Error", "Please select a valid account
type.")

        return

    accounts = load_accounts_from_csv(account_type)

    if validate_account(email, password, accounts):
        messagebox.showinfo("Success", "Sign-in successful!")
    else:
        result = messagebox.askyesno("Account Not Found", "Sign-in
failed. Account not found or invalid credentials. Would you like to create
a new account?")
        if result:
            on_create_account()

    def on_create_account():
        account_type = account_type_var.get()

        if account_type not in ["customer", "retailer"]:
            messagebox.showerror("Error", "Please select a valid account
type.")

        return
```



```
accounts = load_accounts_from_csv(account_type)

new_email = email_entry.get()
new_phone_no = phone_no_entry.get()
new_password = password_entry.get()

if account_type == "customer":
    full_name = full_name_entry.get()
    customer_id = int(customer_id_entry.get())
    customer_factory = CustomerAccountFactory()
    account = customer_factory.create_account()
    account.email = new_email
    account.phone_no = new_phone_no
    account.password = new_password
    account.full_name = full_name
    account.customer_id = customer_id
elif account_type == "retailer":
    retailer_desc = retailer_desc_entry.get()
    retailer_factory = RetailerAccountFactory()
    account = retailer_factory.create_account()
    account.email = new_email
    account.phone_no = new_phone_no
    account.password = new_password
    account.retailer_desc = retailer_desc

save_account_to_csv(account, account_type)
messagebox.showinfo("Account Created",
f"{account_type.capitalize()} account created successfully!")

# GUI Setup
root = tk.Tk()
root.title("Account Management")

# Widgets
account_type_var = tk.StringVar()

tk.Label(root, text="Select Account Type:").grid(row=0, column=0,
padx=10, pady=10)
```

```
tk.Radiobutton(root, text="Customer", variable=account_type_var,
value="customer").grid(row=0, column=1, padx=10, pady=10)

tk.Radiobutton(root, text="Retailer", variable=account_type_var,
value="retailer").grid(row=0, column=2, padx=10, pady=10)

tk.Label(root, text="Email:").grid(row=1, column=0, padx=10, pady=10)
email_entry = tk.Entry(root)
email_entry.grid(row=1, column=1, columnspan=2, padx=10, pady=10)

tk.Label(root, text="Phone No:").grid(row=2, column=0, padx=10,
pady=10)
phone_no_entry = tk.Entry(root)
phone_no_entry.grid(row=2, column=1, columnspan=2, padx=10, pady=10)

tk.Label(root, text="Password:").grid(row=3, column=0, padx=10,
pady=10)
password_entry = tk.Entry(root, show="*")
password_entry.grid(row=3, column=1, columnspan=2, padx=10, pady=10)

tk.Label(root, text="Full Name (Customer):").grid(row=4, column=0,
padx=10, pady=10)
full_name_entry = tk.Entry(root)
full_name_entry.grid(row=4, column=1, columnspan=2, padx=10, pady=10)

tk.Label(root, text="Customer ID (Customer):").grid(row=5, column=0,
padx=10, pady=10)
customer_id_entry = tk.Entry(root)
customer_id_entry.grid(row=5, column=1, columnspan=2, padx=10,
pady=10)

tk.Label(root, text="Retailer Description (Retailer):").grid(row=6,
column=0, padx=10, pady=10)
retailer_desc_entry = tk.Entry(root)
retailer_desc_entry.grid(row=6, column=1, columnspan=2, padx=10,
pady=10)

tk.Button(root, text="Sign In", command=on_sign_in).grid(row=7,
column=0, padx=10, pady=10)
tk.Button(root, text="Create Account",
```

```
command=on_create_account).grid(row=7, column=1, columnspan=2, padx=10,
pady=10)

root.mainloop()

if __name__ == "__main__":
    main_workflow()
```

Output

Based on the figure 6.2 below, there are three images shown. For the first image, it shows the login interface where users can choose account type. If the user account does not exist or is not valid, the second image will appear. Once the user clicks “yes” on the second image, the account will automatically be created as shown in the third image below. This applies to the retailer as well. The only difference is retailer need to input the “Retailer Description (Retailer)” as shown in the first image below.

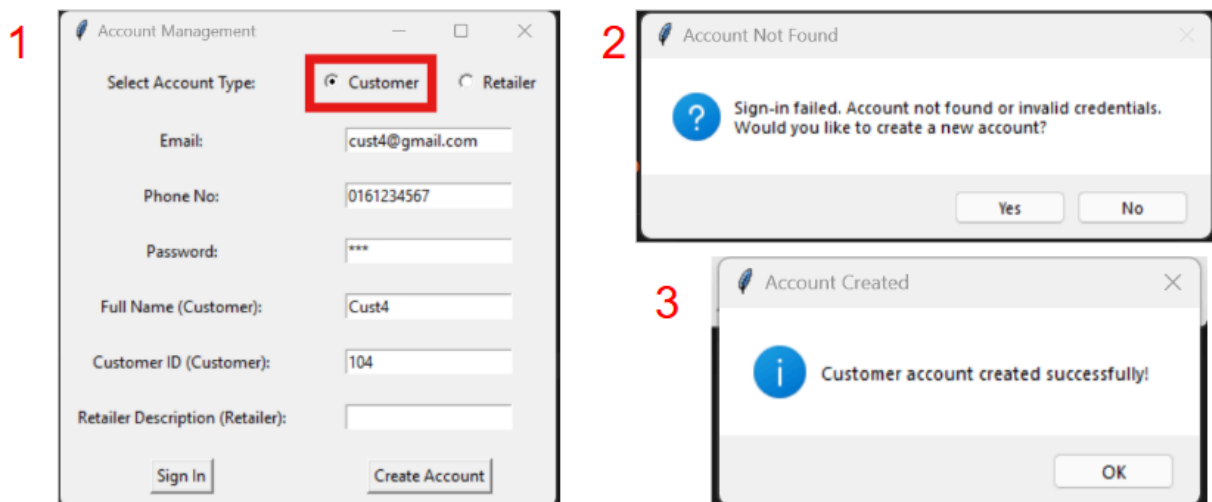


Figure 6.2: Login Page

Figure 6.3 below shows the user has successfully signed in and the data of the registered user account will be stored inside the csv file named (customer_accounts.csv). This action applies for the retailer as well.

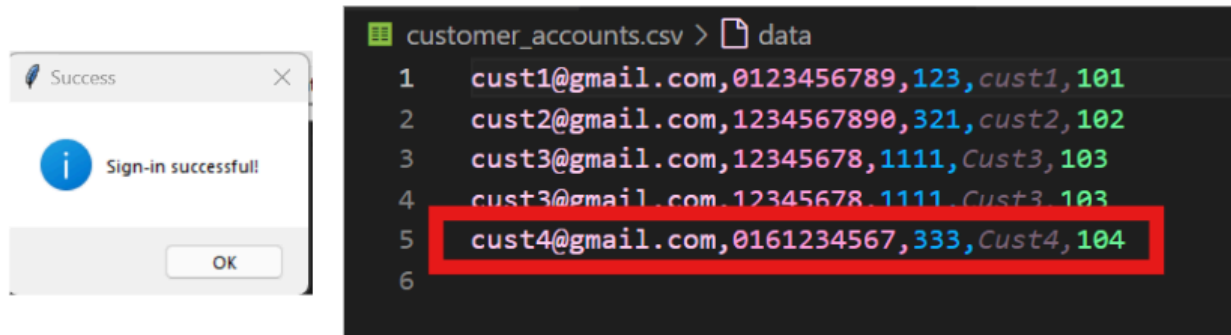


Figure 6.3: Confirmation Page and Database

1.2 Composite (Shopping Cart Management)

Function or Software Component Affected

Implementing a composite pattern for shopping cart allows one to manage complex shopping cart structures where each cart contains products. To add, these products may have other extended features such as options (e.g., color or size). Using this pattern allows treating both individual items and the entire cart as a uniform entity.

Work/Data Flow

1. A ShoppingCart object can contain both individual CartItem objects and other ShoppingCart objects (if nested).
2. Items are added, removed, and calculated using the same interface, whether they are individual or grouped items.

Benefits

- Simplifies the management of cart items and their attributes in a unified way.
- Enables easy nesting of items (e.g., grouping related products) while maintaining simplicity in handling all items as a whole.
- Extends well if additional features are needed (e.g., custom promotions or grouping items).

Limitations

- Overhead can be introduced if cart structures become overly complex.
- May result in performance issues if there are too many nested cart items.

Sample Class Diagram

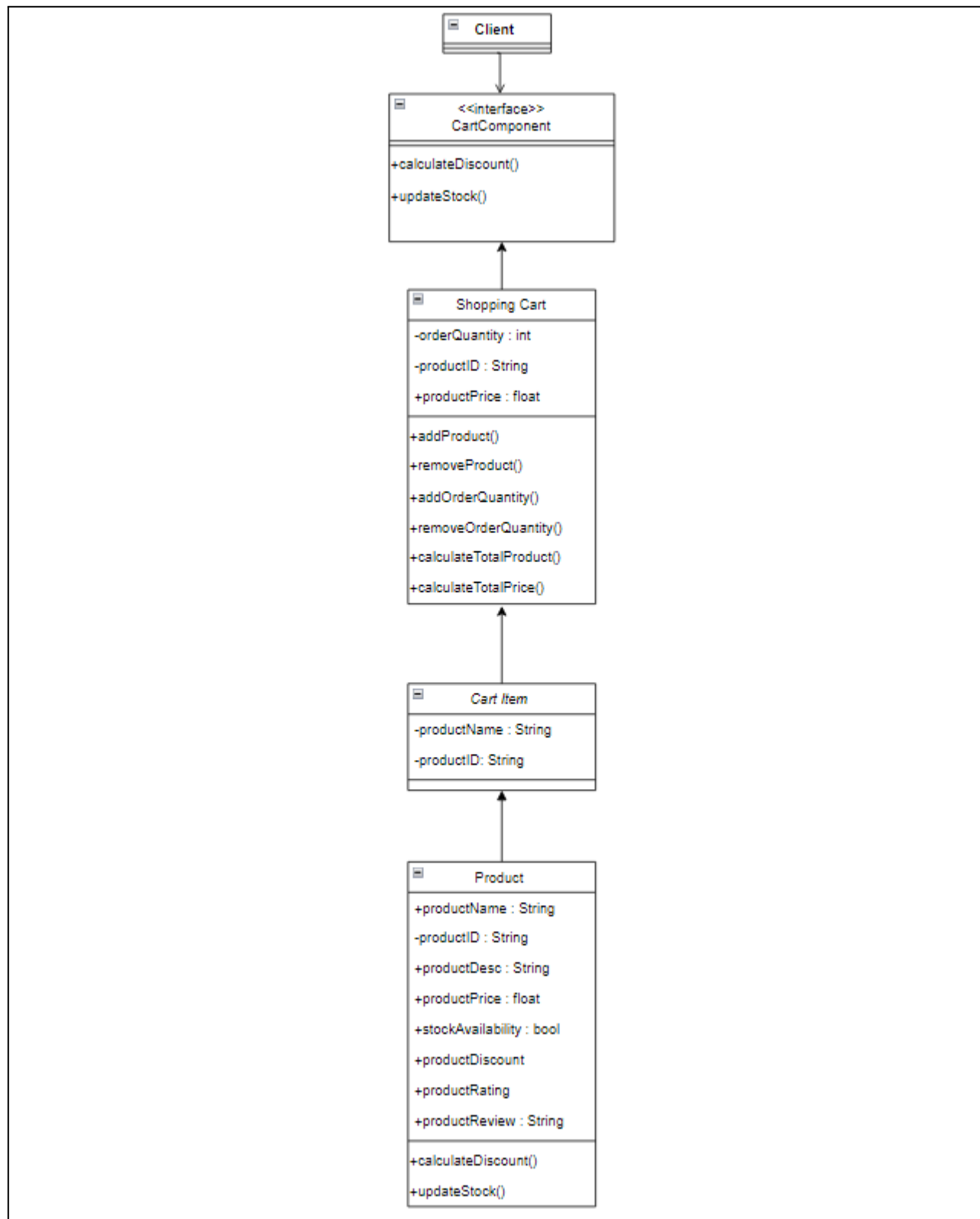


Figure 7.0: Sample Composite Class Diagram

Sample Workflow Diagram

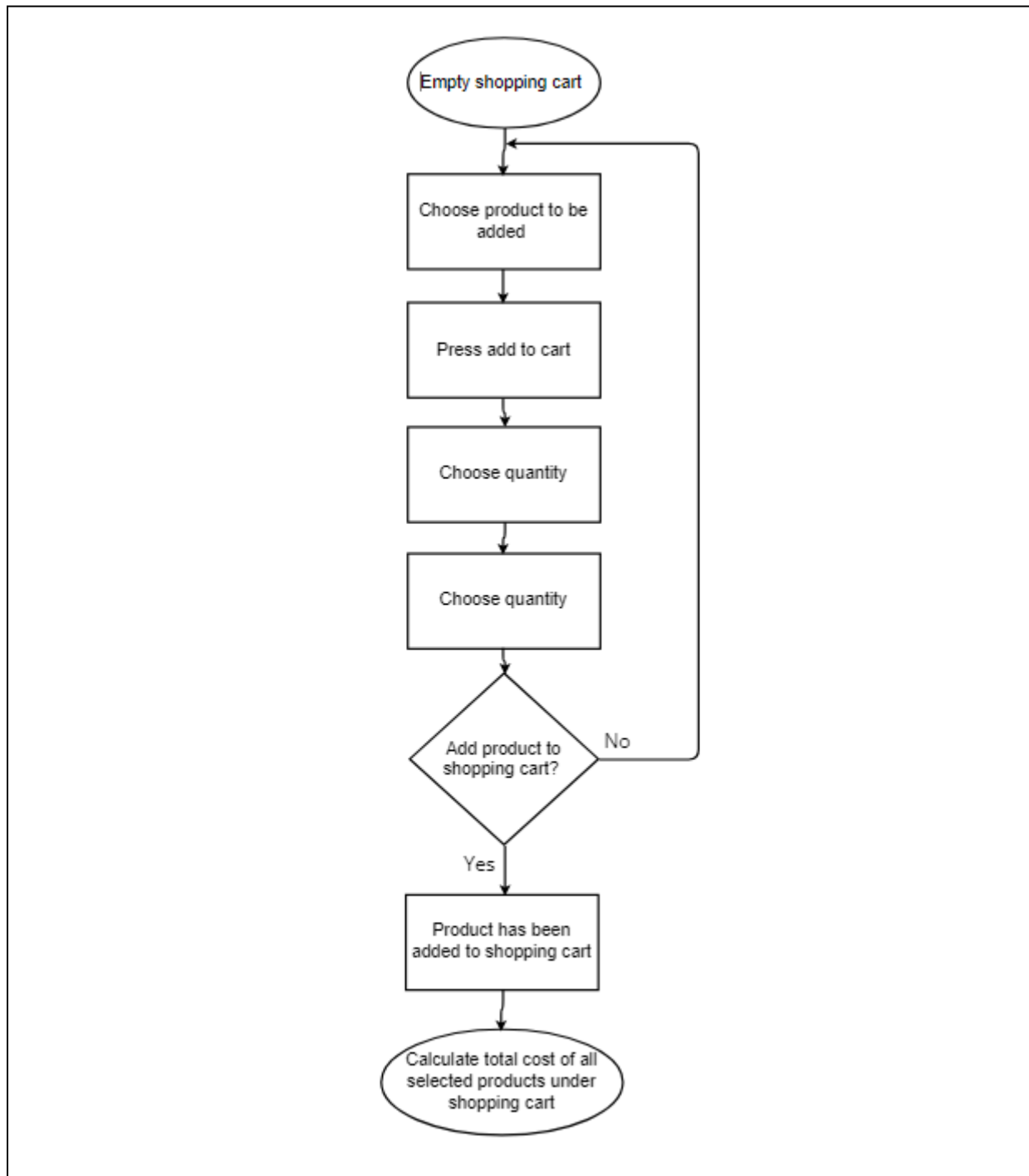


Figure 7.1: Sample Workflow Diagram for Composite

Sample Potential Code

```
import tkinter as tk
from tkinter import messagebox

# Define the Product class
class Product:
    def __init__(self, product_name, product_id, product_desc,
product_price, stock_availability, product_discount, product_rating,
product_review):
        self.product_name = product_name
        self.product_id = product_id
        self.product_desc = product_desc
        self.product_price = product_price
        self.stock_availability = stock_availability
        self.product_discount = product_discount
        self.product_rating = product_rating
        self.product_review = product_review

    def calculate_discount(self):
        # Calculate the discount on the product
        return self.product_price * (self.product_discount / 100)

    def update_stock(self, quantity):
        # Update the stock availability after adding to the cart
        if self.stock_availability >= quantity:
            self.stock_availability -= quantity
            return True
        return False

# Define the CartItem class
class CartItem:
    def __init__(self, product, quantity):
        self.product = product
        self.quantity = quantity

    def calculate_total_price(self):
        # Calculate the total price for this cart item (product price *
```



```
quantity - discount)
    discount = self.product.calculate_discount()
    return (self.product.product_price - discount) * self.quantity

# Define the ShoppingCart class
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_product(self, product, quantity):
        if product.update_stock(quantity):
            self.items.append(CartItem(product, quantity))
            return True
        return False

    def add_cart(self, other_cart):
        self.items.append(other_cart)

    def calculate_total(self):
        total_cost = 0
        for item in self.items:
            if isinstance(item, CartItem):
                total_cost += item.calculate_total_price()
            elif isinstance(item, ShoppingCart):
                total_cost += item.calculate_total()
        return total_cost

# Main GUI class
class ShoppingCartApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Shopping Cart")

        # Initialize the shopping cart
        self.cart = ShoppingCart()

        # Create some products
        self.product1 = Product("Milk", "P001", "Price per liter", 13.00,
100, 10, 4.5, "Farm Fresh.")
```

```
self.product2 = Product("Orange", "P002", "Price per 100g", 1.40,
500, 5, 4.2, "Orange from Australia.")

# Create GUI elements
self.create_widgets()

def create_widgets(self):
    # Product selection
    self.product_label = tk.Label(self.root, text="Select a product to
add:")
    self.product_label.pack()

    self.product_choice = tk.StringVar()
    self.product_choice.set("Milk") # Default choice

    self.milk_radio = tk.Radiobutton(self.root, text="Milk",
variable=self.product_choice, value="Milk", command=self.update_units)
    self.orange_radio = tk.Radiobutton(self.root, text="Orange",
variable=self.product_choice, value="Orange", command=self.update_units)
    self.milk_radio.pack()
    self.orange_radio.pack()

    # Unit selection
    self.unit_label = tk.Label(self.root, text="Select unit:")
    self.unit_label.pack()

    self.unit_var = tk.StringVar()
    self.unit_dropdown = tk.OptionMenu(self.root, self.unit_var, "1
liter", "2 liters", "500 grams", "1000 grams")
    self.unit_var.set("1 liter") # Default unit
    self.unit_dropdown.pack()

    # Add to cart button
    self.add_to_cart_button = tk.Button(self.root, text="Add to Cart",
command=self.add_to_cart)
    self.add_to_cart_button.pack()

    # Calculate total button
    self.calculate_button = tk.Button(self.root, text="Calculate
```

```
Total", command=self.calculate_total)
    self.calculate_button.pack()

    # Shopping cart display
    self.cart_display = tk.Label(self.root, text="Shopping Cart:
Empty")
    self.cart_display.pack()

def update_units(self):
    product_choice = self.product_choice.get()
    self.unit_dropdown['menu'].delete(0, 'end')
    if product_choice == "Milk":
        options = ["1 liter", "2 liters", "3 liters"]
    elif product_choice == "Orange":
        options = ["500 grams", "1000 grams", "1500 grams"]

    for option in options:
        self.unit_dropdown['menu'].add_command(label=option,
command=lambda value=option: self.unit_var.set(value))

    self.unit_var.set(options[0]) # Default to the first option

def add_to_cart(self):
    product_choice = self.product_choice.get()
    selected_unit = self.unit_var.get()

    try:
        if "liter" in selected_unit:
            quantity = float(selected_unit.split()[0]) # Extract
numeric value for liters
        elif "gram" in selected_unit:
            quantity = float(selected_unit.split()[0]) / 100 #
Convert grams to 100g units
        else:
            raise ValueError("Invalid unit selected")

    if product_choice == "Milk":
        selected_product = self.product1
    elif product_choice == "Orange":
```

```
        selected_product = self.product2
    else:
        messagebox.showerror("Error", "Invalid product choice!")
        return

    # Try to add the product to the cart
    if self.cart.add_product(selected_product, quantity):
        self.cart_display.config(text=f"Shopping Cart: Added
{selected_unit} of {selected_product.product_name}")
        messagebox.showinfo("Success",
f"{selected_product.product_name} has been added to the cart.")
    else:
        messagebox.showerror("Error", "Insufficient stock!")
    except ValueError:
        messagebox.showerror("Error", "Please select a valid unit.")

    def calculate_total(self):
        total_cost = self.cart.calculate_total()
        messagebox.showinfo("Total Cost", f"The total cost of the shopping
cart is: ${total_cost:.2f}")

# Create the main application window
root = tk.Tk()
app = ShoppingCartApp(root)
root.mainloop()
```

Output

Figure 7.2 shows the shopping cart page where customers can select products they want to add in the cart and unit based on type of products. In the first page below, it shows milk products and the unit in “liter” while on the second image, it shows “grams” for the product of oranges.

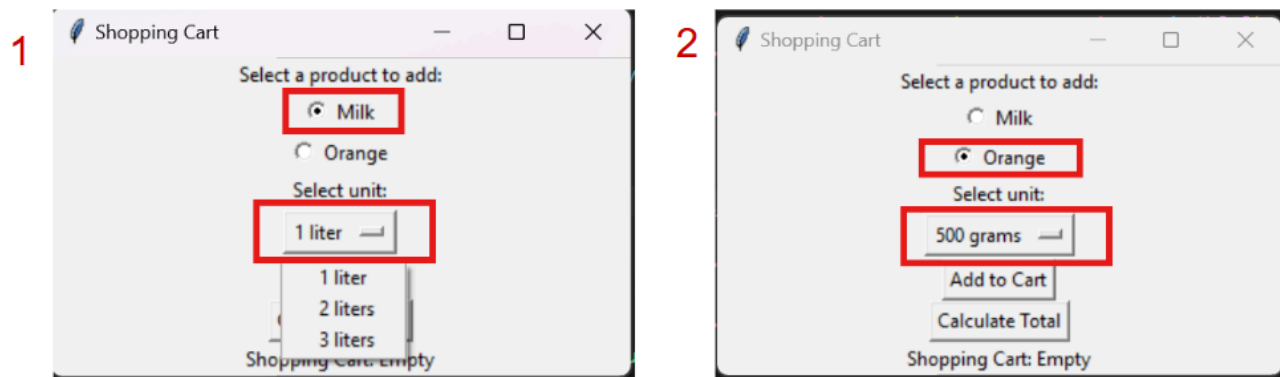


Figure 7.2: Shopping Cart Page

After a customer adds a product to cart by clicking the “Add to Cart” button, the successful message will appear as shown in the second image on figure 7.3 below. The customer calculates the total and the message will appear by showing the total cost of the product that has been added in the cart.

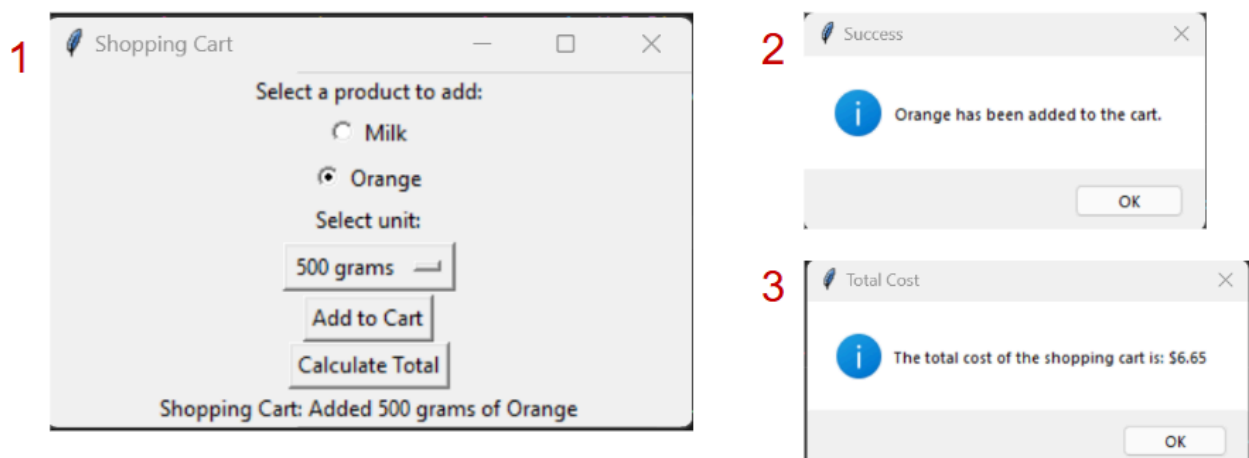


Figure 7.3: Successful and Total Cost Page

1.3 Observer (Notifications, Product Stock & Management)

Function or Software Component Affected

- Handling real-time notifications, such as sending updates on order status or inventory changes. Observers (e.g., UI elements, backend systems) are notified when there are changes to the subject (e.g., order status or stock levels).
- Tracking changes in product stock levels and notifying relevant systems (e.g., updating the product display, alerting when products are low in stock). The Observer pattern is ideal for these real-time updates.

Work/Data Flow (Notifications)

1. The NotificationService (subject) maintains a list of subscribers (observers) who are interested in specific updates.
2. When a significant event occurs (e.g., stock change, order update), it notifies all registered observers about the change.
3. Observers update themselves (e.g., updating UI components or sending emails).

Work/Data Flow (Product Stock & Management)

1. The Product class acts as the subject that maintains a list of observers (e.g., the inventory system or UI).
2. When stock levels change (e.g., when an item is bought), the Product notifies all observers about the stock change.
3. Observers respond by updating their state accordingly (e.g., adjusting available stock or triggering restocking).

Benefits (Notifications)

- Decouples the notification system from other components, making it flexible and easy to extend.
- Enables real-time updates with minimal delay.
- Allows multiple observers to receive notifications without the need for direct coupling.

Benefits (Product Stock & Management)

- Enables real-time synchronization between product availability and other systems.
- Decouples stock management from other system components, making it easy to extend and maintain.
- Allows multiple systems to react independently to stock changes.

Limitations (Notifications)

- Subscription management can become complex if many observers are involved.
- Memory leaks can occur if observers are not properly unsubscribed after use.

Limitations (Product Stock & Management)

- Overhead in managing many subscribers if the number of product observers is large.
- Potential memory leaks if observers are not unsubscribed correctly.

Sample Class Diagrams

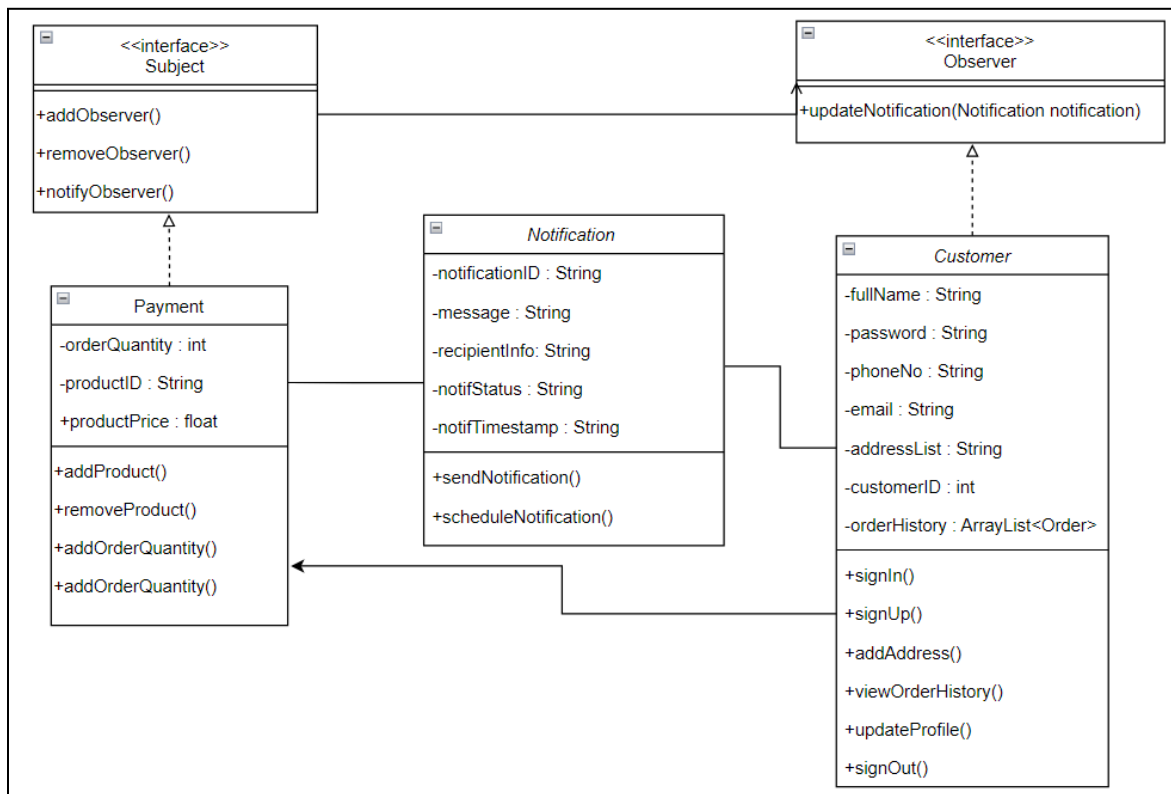


Figure 8.0: Sample Observer Class Diagram (Notifications)

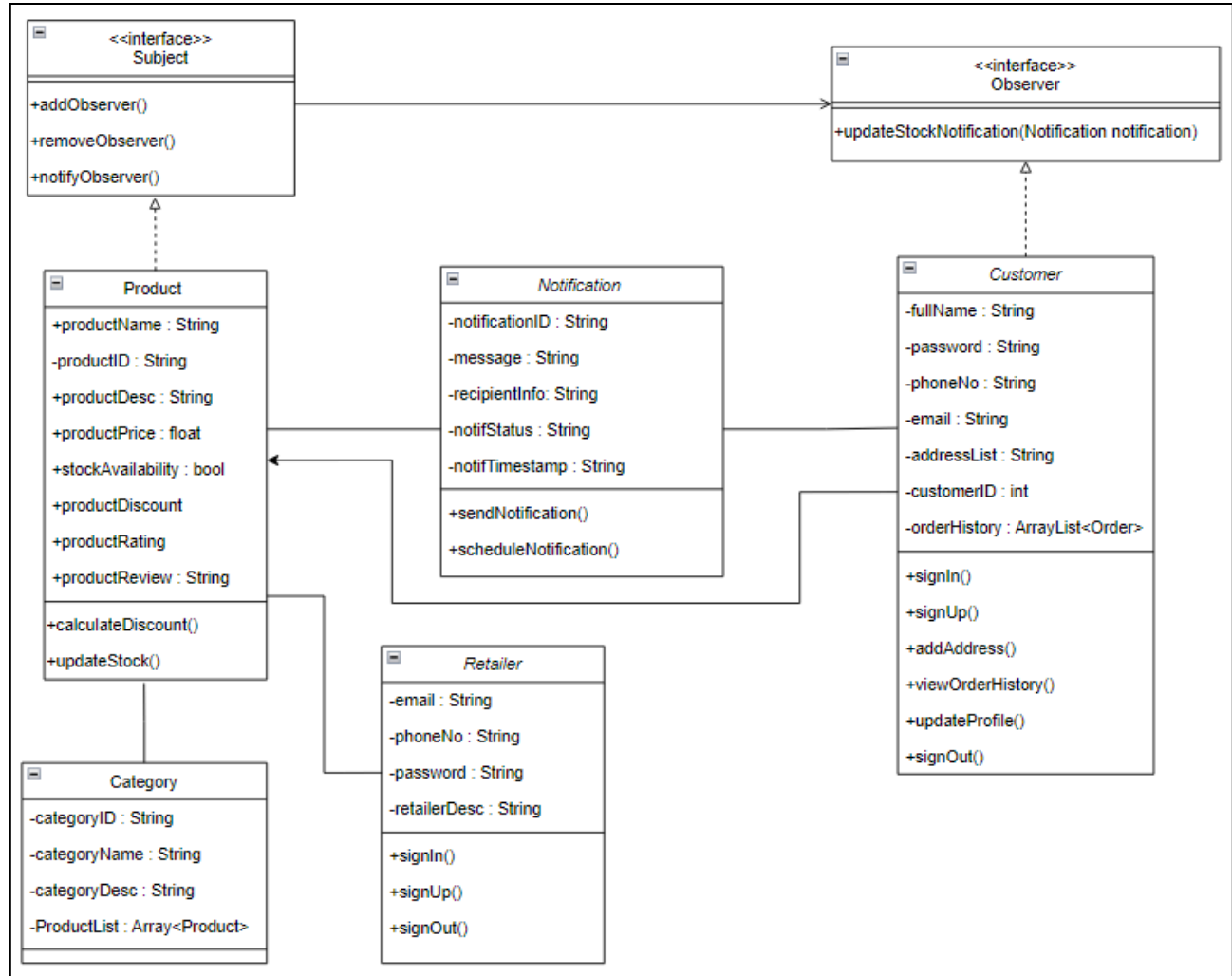


Figure 8.1: Sample Observer Class Diagram (Product Stock & Management)

Sample Workflow Diagram

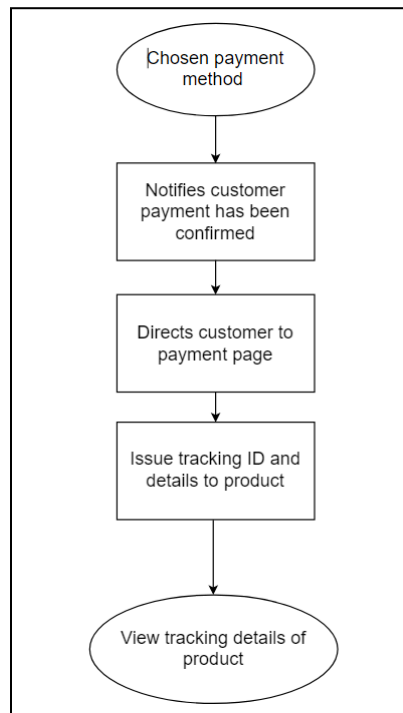


Figure 8.2: Sample Workflow Diagram for Observer (Notification)

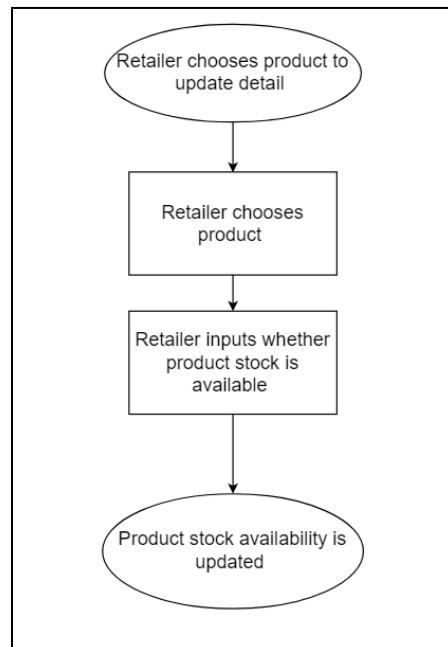


Figure 8.3: Sample Workflow Diagram for Observer (Product Stock & Management)
Sample Potential Code

```
import tkinter as tk
from tkinter import messagebox

# Observer pattern implementation
class Subject:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        self._observers.append(observer)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, notification):
        for observer in self._observers:
            observer.update_notification(notification)

class Observer:
    def update_notification(self, notification):
        pass

class Notification:
    def __init__(self, notification_id, message, recipient_info):
        self.notification_id = notification_id
        self.message = message
        self.recipient_info = recipient_info

class Payment(Subject):
    def __init__(self):
        super().__init__()
        self.order_quantity = 0
        self.product_id = ""
```

```
        self.product_price = 0.0

    def confirm_payment(self, customer):
        notification = Notification("001", "Payment Confirmed",
customer.email)
        self.notify_observers(notification)

class Customer(Observer):
    def __init__(self, full_name, email):
        self.full_name = full_name
        self.email = email
        self.tracking_details = None

    def update_notification(self, notification):
        messagebox.showinfo("Notification", f"{notification.message} for
{notification.recipient_info}")

    def view_tracking_details(self):
        if self.tracking_details:
            messagebox.showinfo("Tracking Details", self.tracking_details)
        else:
            messagebox.showwarning("Tracking Details", "No tracking
details available.")

# GUI implementation
class Application:
    def __init__(self, root):
        self.root = root
        self.root.title("Observer Pattern - Payment System")

        # Customer setup
        self.customer = Customer("Cust 4", "cust4@gmail.com")
        self.payment = Payment()
        self.payment.add_observer(self.customer)

        # GUI Components
        tk.Label(root, text="Payment System", font=("Arial",
```

```
16)).pack(pady=10)

    tk.Label(root, text="Choose Payment Method:").pack()
    self.payment_method = tk.StringVar(value="Credit Card")
    tk.OptionMenu(root, self.payment_method, "Credit Card", "PayPal",
"Bank Transfer").pack(pady=5)

    tk.Button(root, text="Confirm Payment",
command=self.confirm_payment).pack(pady=10)
    tk.Button(root, text="View Tracking Details",
command=self.view_tracking).pack(pady=10)

    def confirm_payment(self):
        payment_method = self.payment_method.get()
        self.payment.confirm_payment(self.customer)
        self.customer.tracking_details = f"Tracking ID: TRK12345\nProduct
ID: P001\nPayment Method: {payment_method}"
        messagebox.showinfo("Payment Page", "Redirecting to payment
page...")

    def view_tracking(self):
        self.customer.view_tracking_details()

if __name__ == "__main__":
    root = tk.Tk()
    app = Application(root)
    root.mainloop()
```

Output

Based on figure 8.4 shown below, customers can choose the payment method shown in the first image. After the confirmation payment has been clicked by customer, the notification will appear to notify the user that the payment has been confirmed as shown in the third image and customer can view the tracking details by clicking “View Tracking Details” and the tracking page will appear as shown in the fourth image below.

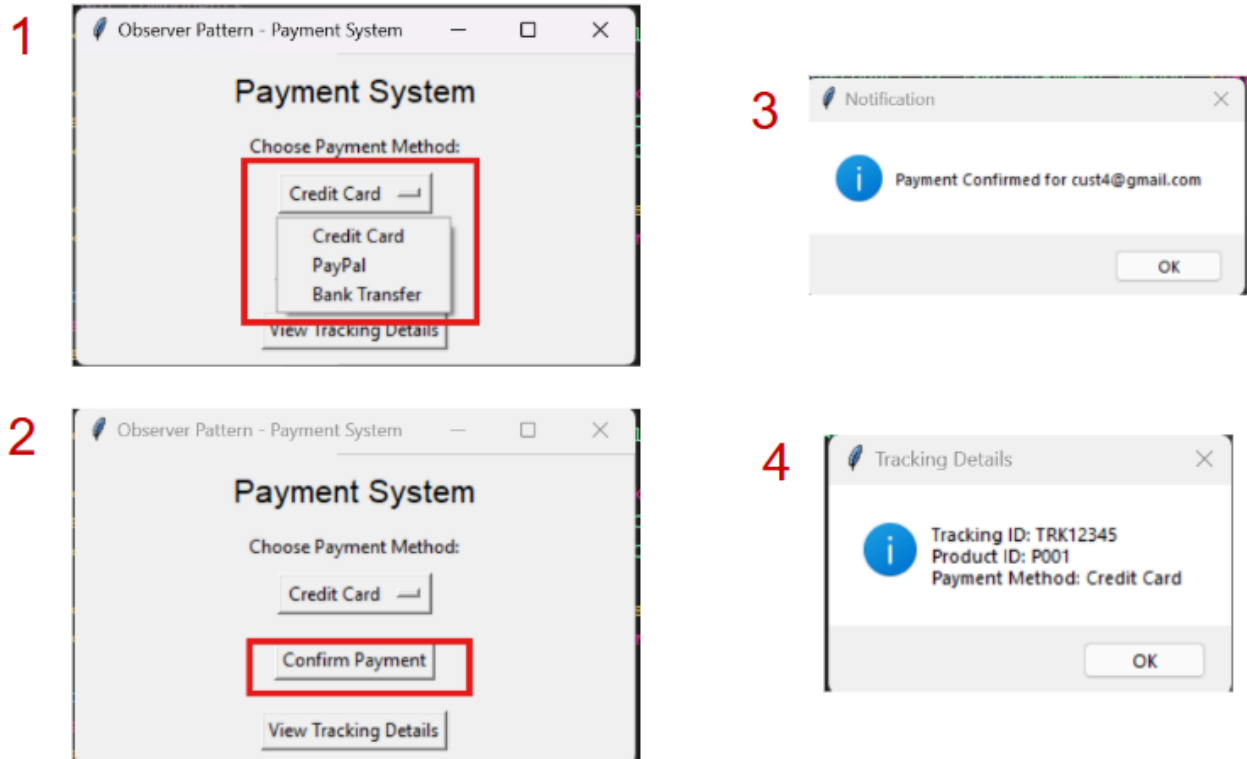


Figure 8.4: Sample Output for Observer (Notification)

Sample Potential Code

```
import tkinter as tk
from tkinter import messagebox

# Define the Product class (without the discount logic)
class Product:
    def __init__(self, product_name, product_id, product_desc,
product_price, stock_availability, product_rating, product_review):
        self.product_name = product_name
        self.product_id = product_id
        self.product_desc = product_desc
        self.product_price = product_price
        self.stock_availability = stock_availability
        self.product_rating = product_rating
        self.product_review = product_review

    def update_stock(self, quantity):
        # Update the stock availability
        self.stock_availability += quantity

    def reduce_stock(self, quantity):
        # Reduce stock after adding to the cart
        if self.stock_availability >= quantity:
            self.stock_availability -= quantity
            return True
        return False

# Define the CartItem class
class CartItem:
    def __init__(self, product, quantity):
        self.product = product
        self.quantity = quantity

    def calculate_total_price(self):
        # Calculate the total price for this cart item (product price *
quantity)
        return self.product.product_price * self.quantity
```

```
# Define the ShoppingCart class
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_product(self, product, quantity):
        if product.reduce_stock(quantity):
            self.items.append(CartItem(product, quantity))
            return True
        return False

    def calculate_total(self):
        total_cost = 0
        for item in self.items:
            total_cost += item.calculate_total_price()
        return total_cost

# Main GUI class
class ShoppingCartApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Shopping Cart")

        # Initialize the shopping cart
        self.cart = ShoppingCart()

        # Create some products
        self.product1 = Product("Milk", "P001", "Price per liter", 13.00,
100, 4.5, "Farm Fresh.")
        self.product2 = Product("Orange", "P002", "Price per 100g", 1.40,
500, 4.2, "Orange from Australia.")

        # Create GUI elements
        self.create_widgets()

    def create_widgets(self):
        # Product selection
        self.product_label = tk.Label(self.root, text="Select a product to
update details:")
```

```
self.product_label.pack()

self.product_choice = tk.StringVar()
self.product_choice.set("Milk") # Default choice

self.product_dropdown = tk.OptionMenu(self.root,
self.product_choice, "Milk", "Orange", command=self.update_units)
self.product_dropdown.pack()

# Stock availability input
self.stock_label = tk.Label(self.root, text="Enter product stock
availability:")
self.stock_label.pack()

# Unit selection (default option for milk and orange)
self.unit_var = tk.StringVar()
self.unit_dropdown = tk.OptionMenu(self.root, self.unit_var, "1
liter", "2 liters", "500 grams", "1000 grams")
self.unit_var.set("1 liter") # Default unit
self.unit_dropdown.pack()

# Update stock button
self.update_stock_button = tk.Button(self.root, text="Update
Stock", command=self.update_stock)
self.update_stock_button.pack()

# Stock display
self.stock_display = tk.Label(self.root, text="Stock: Milk - 100
liters, Orange - 500 grams")
self.stock_display.pack()

def update_units(self, *args):
    """Updates the available units based on selected product"""
    product_choice = self.product_choice.get()
    self.unit_dropdown['menu'].delete(0, 'end') # Clear current
options

    if product_choice == "Milk":
        options = ["1 liter", "2 liters", "3 liters"]
```



```
elif product_choice == "Orange":
    options = ["500 grams", "1000 grams", "1500 grams"]

    # Add new options based on the selected product
    for option in options:
        self.unit_dropdown['menu'].add_command(label=option,
command=lambda value=option: self.unit_var.set(value))

    self.unit_var.set(options[0]) # Default to the first option

def update_stock(self):
    """Handles stock update based on selected product and quantity"""
    product_choice = self.product_choice.get()
    selected_quantity = self.unit_var.get()

    try:
        if "liter" in selected_quantity:
            quantity = int(selected_quantity.split()[0]) # Extract
numeric value for liters
        elif "gram" in selected_quantity:
            quantity = int(selected_quantity.split()[0]) / 100 #
Convert grams to 100g units
        else:
            raise ValueError("Invalid unit selected")

        # Update stock for the selected product
        if product_choice == "Milk":
            self.product1.update_stock(quantity)
        elif product_choice == "Orange":
            self.product2.update_stock(quantity)
        else:
            messagebox.showerror("Error", "Invalid product choice!")
            return

        # Update stock display
        self.stock_display.config(text=f"Stock: Milk -
{self.product1.stock_availability} liters, Orange -
{int(self.product2.stock_availability * 100)} grams")
        messagebox.showinfo("Success", f"Stock for {product_choice}")
```

```

updated successfully!")
    except ValueError:
        messagebox.showerror("Error", "Please select a valid
quantity.")

# Create the main application window
root = tk.Tk()
app = ShoppingCartApp(root)
root.mainloop()

```

Output

In the first image shown in figure 8.5 below, it shows the current product stock. Retailers can update the product stock by selecting a product and the availability as shown in the second image below. After retailers update the product stock, the successful message will appear to notify retailers that the stock has been successfully updated as shown in image 3 below. Then, it will update the stock as shown in image 4 below.

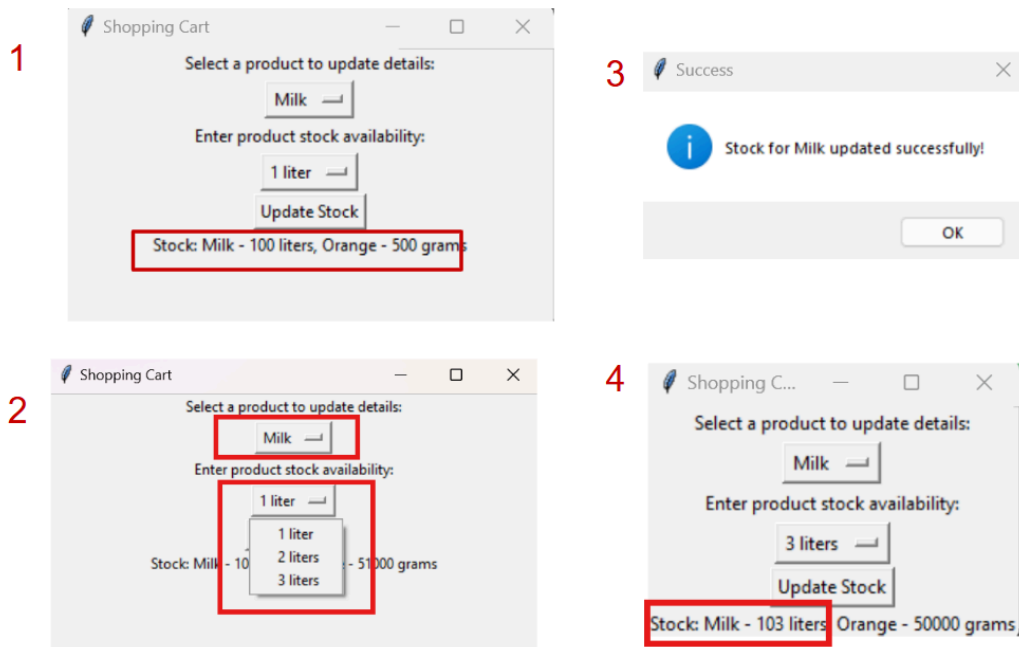


Figure 8.5: Sample Output for Observer (Product Stock & Management)

1.4 Strategy (Order & Payments)

Function or Software Component Affected

Handling different payment methods (e.g., credit card, PayPal, cash) where each payment method follows a distinct process. The Strategy pattern allows switching between payment methods dynamically based on the user's choice.

Work/Data Flow

1. The client selects a payment method.
2. A PaymentStrategy interface is defined, and concrete payment strategies are implemented (e.g., CreditCardPayment, PaypalPayment).
3. The Order object delegates the payment processing to the appropriate strategy, which then processes the payment based on the chosen method.

Benefits

- Flexibility to add new payment methods without modifying the **Order** class.
- Centralized management of different payment strategies.
- Reduces the need for complex conditional logic for handling payments.

Limitations

- Introduces additional classes (payment strategies) for each payment method, potentially making the system more complex.
- Requires changes in existing code when adding a new payment method.

Sample Class Diagram

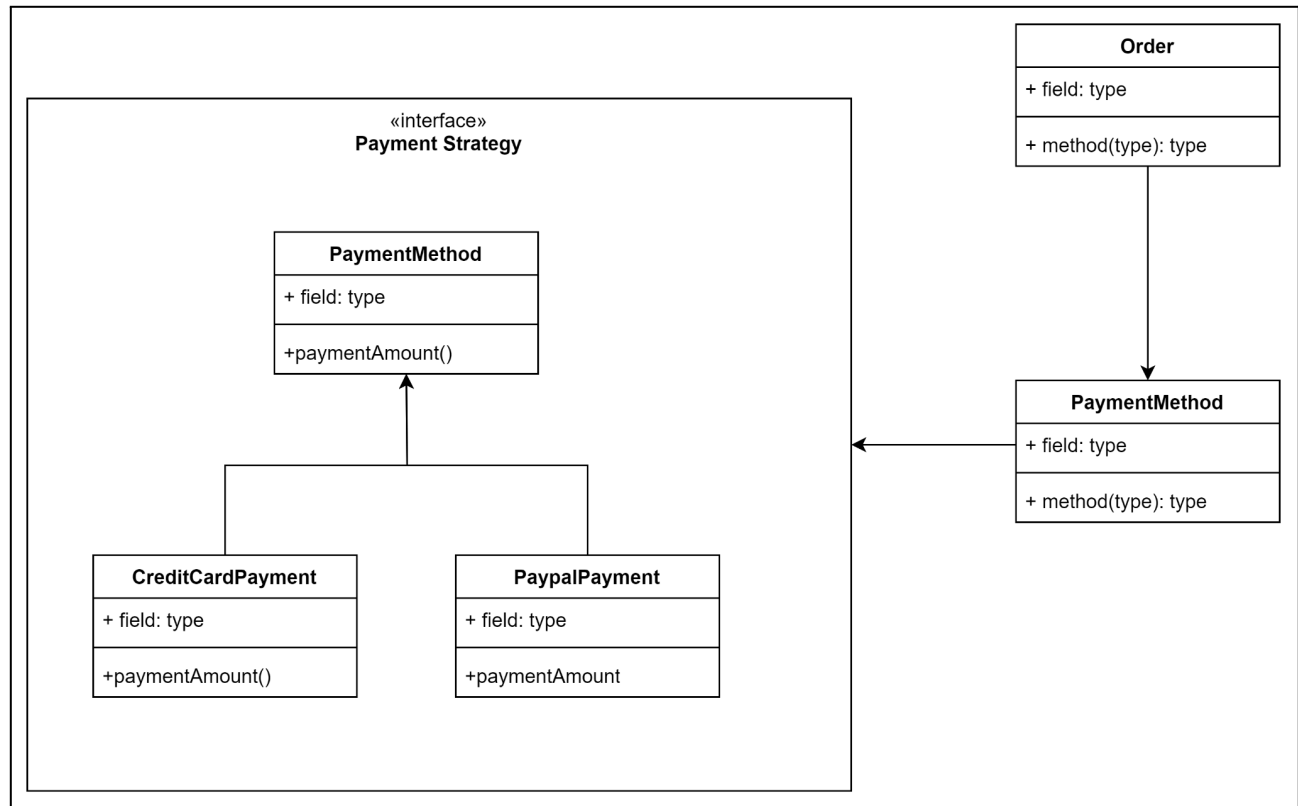


Figure 9.0: Sample Strategy Class Diagram

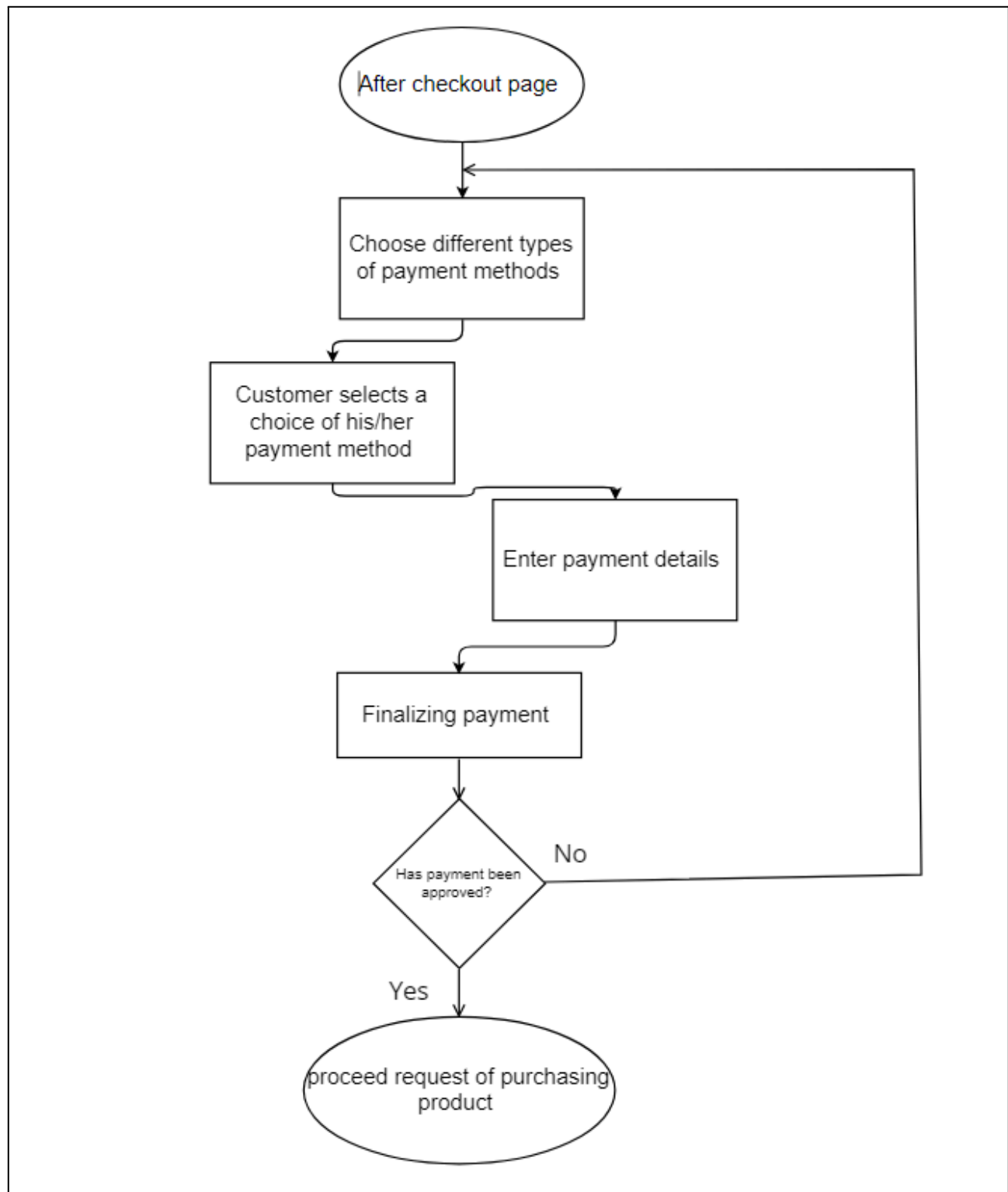
Sample Workflow Diagram

Figure 9.1: Sample Workflow Diagram for Strategy

Sample Potential Code

```
import tkinter as tk
from tkinter import messagebox

# Base class for payment methods
class PaymentMethod:
    def __init__(self):
        self.amount = 0.0

    def payment_amount(self):
        pass

# Credit Card Payment class
class CreditCardPayment(PaymentMethod):
    def __init__(self):
        super().__init__()
        self.card_number = ""
        self.expiry_date = ""
        self.cvv = ""

    def payment_amount(self):
        return self.amount

# PayPal Payment class
class PaypalPayment(PaymentMethod):
    def __init__(self):
        super().__init__()
        self.paypal_email = ""

    def payment_amount(self):
        return self.amount

# GUI Implementation
class Application:
    def __init__(self, root):
        self.root = root
        self.root.title("Payment System")
```

```
# Initialize payment methods
self.credit_card_payment = CreditCardPayment()
self.paypal_payment = PaypalPayment()
self.selected_payment_method = None

# GUI Components
tk.Label(root, text="Checkout Page", font=("Arial",
16)).pack(pady=10)

tk.Label(root, text="Choose Payment Method:").pack()
self.payment_method = tk.StringVar(value="Credit Card")
tk.OptionMenu(root, self.payment_method, "Credit Card",
"PayPal").pack(pady=5)

tk.Button(root, text="Select Payment Method",
command=self.select_payment_method).pack(pady=10)

self.payment_frame = tk.Frame(root)
self.payment_frame.pack(pady=10)

self.finalize_button = tk.Button(root, text="Finalize Payment",
command=self.finalize_payment, state=tk.DISABLED)
self.finalize_button.pack(pady=10)

def select_payment_method(self):
    for widget in self.payment_frame.winfo_children():
        widget.destroy()

    selected_method = self.payment_method.get()
    if selected_method == "Credit Card":
        self.selected_payment_method = self.credit_card_payment
        tk.Label(self.payment_frame, text="Enter Credit Card
Details:").pack()
        tk.Label(self.payment_frame, text="Card Number:").pack()
        self.card_number_entry = tk.Entry(self.payment_frame)
        self.card_number_entry.pack()
        tk.Label(self.payment_frame, text="Expiry Date
(MM/YY):").pack()
```

```
        self.expiry_date_entry = tk.Entry(self.payment_frame)
        self.expiry_date_entry.pack()
        tk.Label(self.payment_frame, text="CVV:").pack()
        self.cvv_entry = tk.Entry(self.payment_frame, show="*")
        self.cvv_entry.pack()

    elif selected_method == "PayPal":
        self.selected_payment_method = self.paypal_payment
        tk.Label(self.payment_frame, text="Enter PayPal
Email:").pack()

        self.paypal_email_entry = tk.Entry(self.payment_frame)
        self.paypal_email_entry.pack()

    self.finalize_button.config(state=tk.NORMAL)

    def finalize_payment(self):
        selected_method = self.payment_method.get()
        if selected_method == "Credit Card":
            self.selected_payment_method.card_number =
self.card_number_entry.get()
            self.selected_payment_method.expiry_date =
self.expiry_date_entry.get()
            self.selected_payment_method.cvv = self.cvv_entry.get()
            if not self.selected_payment_method.card_number or not
self.selected_payment_method.expiry_date or not
self.selected_payment_method.cvv:
                messagebox.showwarning("Error", "Please enter all credit
card details.")
            return
        elif selected_method == "PayPal":
            self.selected_payment_method.paypal_email =
self.paypal_email_entry.get()
            if not self.selected_payment_method.paypal_email:
                messagebox.showwarning("Error", "Please enter your PayPal
email.")
            return

        if messagebox.askyesno("Payment Confirmation", "Has the payment
been approved?"):
            messagebox.showinfo("Success", "Payment finalized. Proceeding
```



```
to purchase product.")
    else:
        messagebox.showinfo("Retry", "Returning to payment method
selection.")
        self.select_payment_method()

if __name__ == "__main__":
    root = tk.Tk()
    app = Application(root)
    root.mainloop()
```

Output

Figure 9.2 shows the payment page where customers can choose payment methods as shown in the first image below. Customers will input the data based on the payment method that has been selected as shown in the second and third image below. .

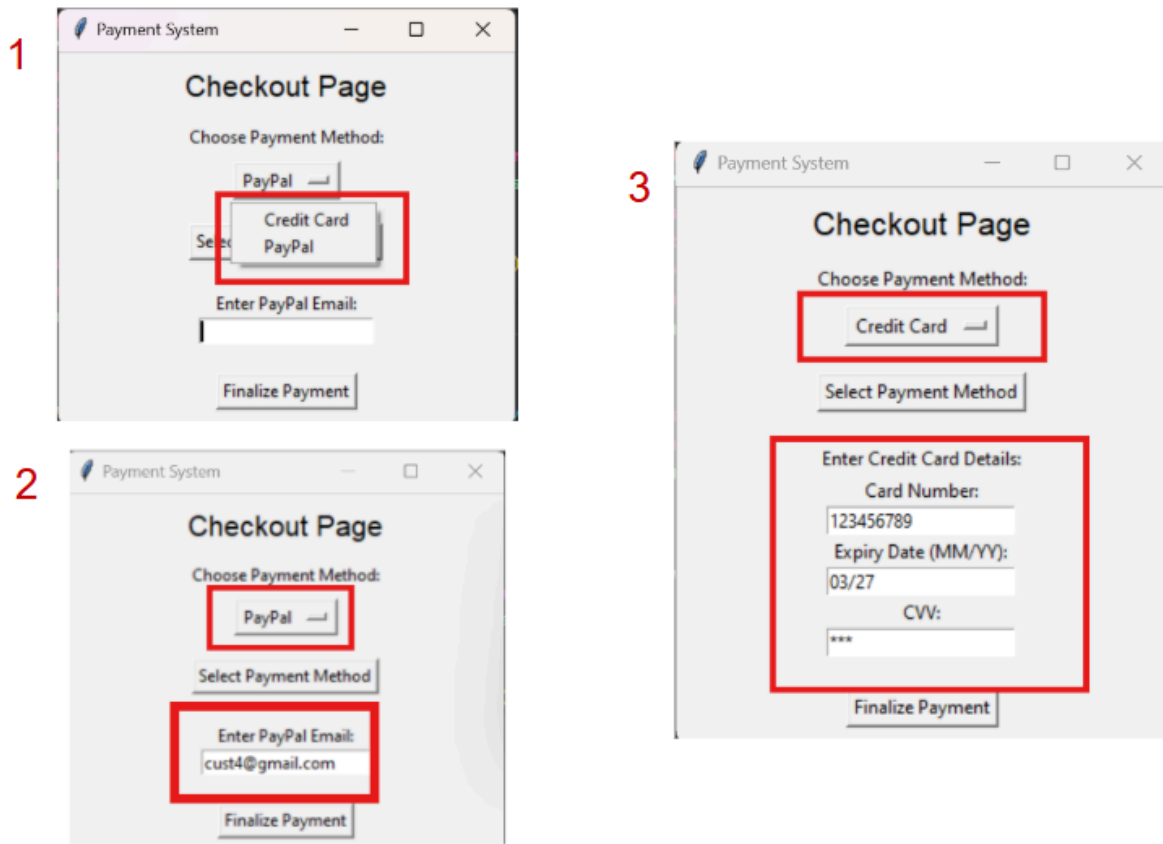


Figure 9.2: Payment Process Page

In the first image shown in figure 9.3 below, it shows a confirmation message. When the customer agrees with the payment by clicking “yes” button, the successful page will appear as shown in the second image below and it will return to the payment page as shown in image 3 below.

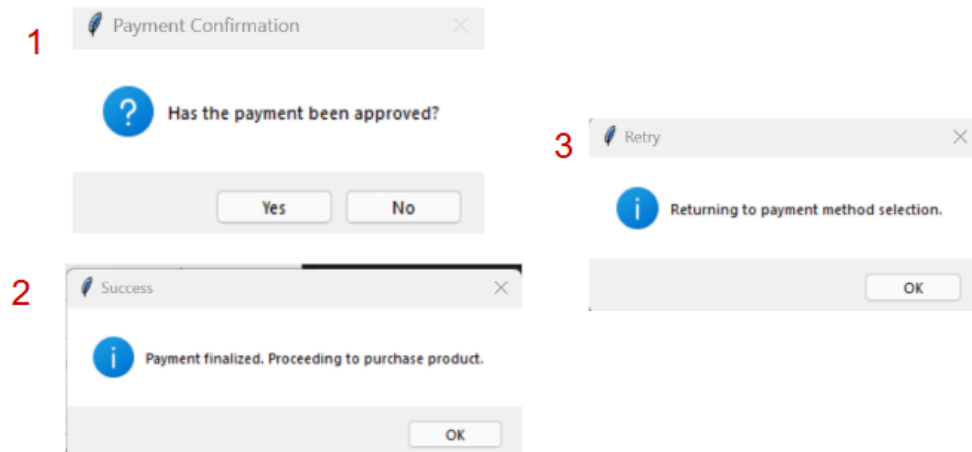


Figure 9.3: Confirmation Page

1.5 Builder (Order and Payment)

Function or Software Component Affected

Generating complex receipts with various parts, such as product details, taxes, shipping costs, and discounts. The Builder pattern allows constructing the payment receipt and the order ticket step by step.

Work/Data Flow

1. An OrderBuilder class defines the step-by-step process for constructing a receipt.
2. The builder adds details progressively, such as product prices, tax calculations, shipping costs, and final totals.
3. Once all components are added, the builder returns the complete Receipt object.

Benefits

- Simplifies the creation of complex receipt objects with multiple components.
- Provides flexibility by allowing the construction process to be customized.
- Reduces code complexity by separating the construction logic from the actual receipt object.

Limitations

- Requires additional classes to define the builder and its steps.
- If not carefully structured, it can lead to over-engineering for simple receipt requirements.

Sample Class Diagram

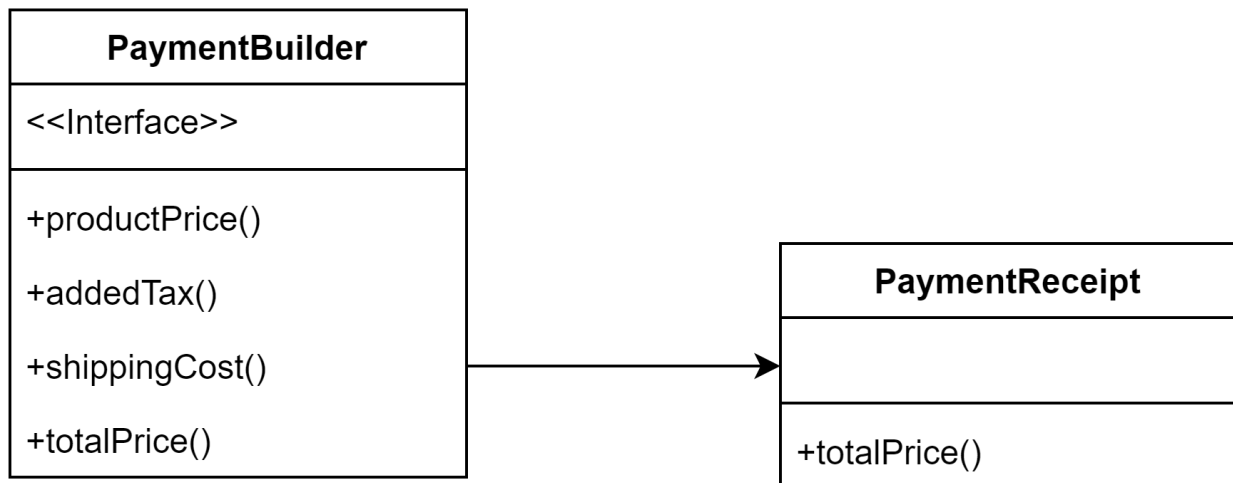


Figure 10.0: Sample Builder Class Diagram

Sample Workflow Diagram

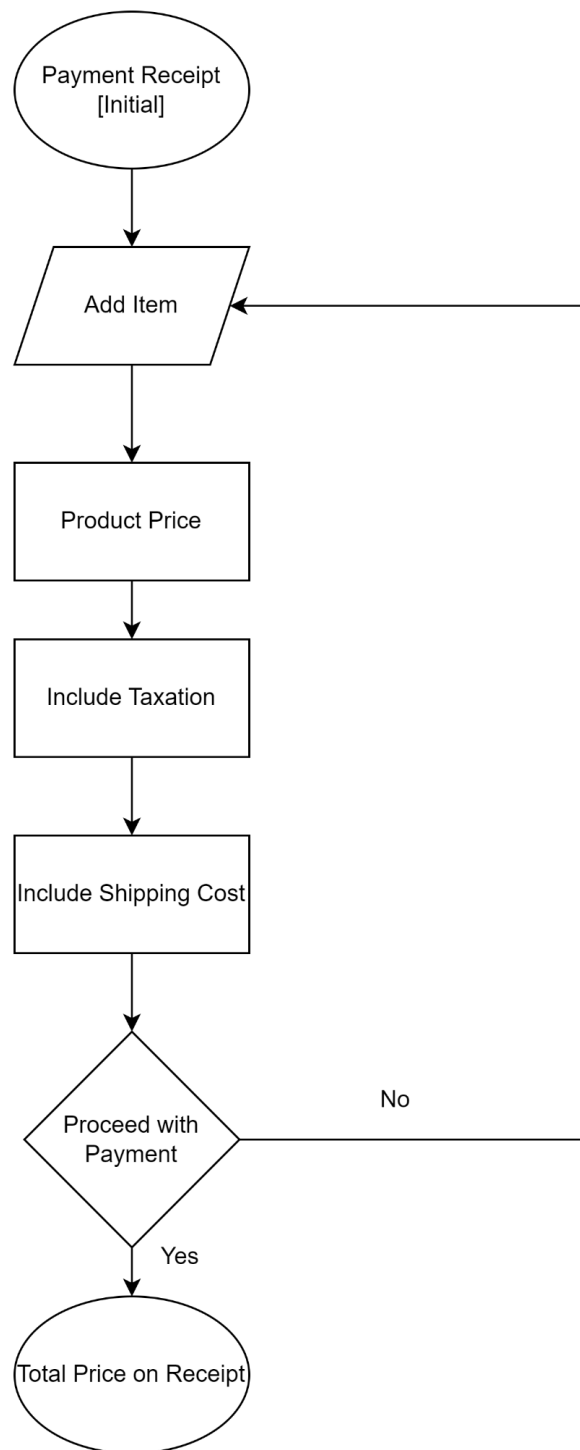


Figure 10.1: Sample Workflow Diagram for Builder

Sample Potential Code

```
from abc import ABC, abstractmethod
import tkinter as tk
from tkinter import messagebox

# Abstract Interface for PaymentBuilder
class PaymentBuilder(ABC):
    @abstractmethod
    def productPrice(self):
        pass

    @abstractmethod
    def addedTax(self):
        pass

    @abstractmethod
    def shippingCost(self):
        pass

    @abstractmethod
    def totalPrice(self):
        pass

# Concrete Implementation of PaymentReceipt
class PaymentReceipt(PaymentBuilder):
    def __init__(self, product_price, tax_rate, shipping_fee):
        self.product_price = product_price
        self.tax_rate = tax_rate
        self.shipping_fee = shipping_fee

    def productPrice(self):
        return self.product_price

    def addedTax(self):
        return self.product_price * self.tax_rate

    def shippingCost(self):
```

```
        return self.shipping_fee

    def totalPrice(self):
        return self.productPrice() + self.addedTax() + self.shippingCost()

# GUI for Payment Builder
class PaymentBuilderGUI:
    def __init__(self, root):
        self.root = root

        self.root.title("Payment Builder")
        self.root.geometry("400x300")

        # Input fields
        tk.Label(root, text="Product Price:", font=("Helvetica",
12)).grid(row=0, column=0, sticky="e", padx=10, pady=5)
        self.product_price_entry = tk.Entry(root, font=("Helvetica", 12))
        self.product_price_entry.grid(row=0, column=1, padx=10, pady=5)

        tk.Label(root, text="Tax Rate (as decimal):", font=("Helvetica",
12)).grid(row=1, column=0, sticky="e", padx=10, pady=5)
        self.tax_rate_entry = tk.Entry(root, font=("Helvetica", 12))
        self.tax_rate_entry.grid(row=1, column=1, padx=10, pady=5)

        tk.Label(root, text="Shipping Fee:", font=("Helvetica",
12)).grid(row=2, column=0, sticky="e", padx=10, pady=5)
        self.shipping_fee_entry = tk.Entry(root, font=("Helvetica", 12))
        self.shipping_fee_entry.grid(row=2, column=1, padx=10, pady=5)

        # Calculate button
        tk.Button(root, text="Calculate Total Price",
command=self.calculate_total_price, font=("Helvetica", 12), bg="#4CAF50",
fg="white").grid(row=3, column=0, columnspan=2, pady=10)

        # Print button
        tk.Button(root, text="Print Receipt", command=self.print_receipt,
font=("Helvetica", 12), bg="#008CBA", fg="white").grid(row=4, column=0,
columnspan=2, pady=10)

        # Result label
```

```
self.result_label = tk.Label(root, text="", font=("Helvetica",
12))

self.result_label.grid(row=5, column=0, columnspan=2)

def calculate_total_price(self):
    try:
        product_price = float(self.product_price_entry.get())
        tax_rate = float(self.tax_rate_entry.get())
        shipping_fee = float(self.shipping_fee_entry.get())

        # Create PaymentReceipt object
        self.receipt = PaymentReceipt(product_price, tax_rate,
shipping_fee)
        total = self.receipt.totalPrice()

        self.result_label.config(text=f"Total Price: ${total:.2f}")
    except ValueError:
        messagebox.showerror("Invalid Input", "Please enter valid
numbers for all fields.")

def print_receipt(self):
    if hasattr(self, 'receipt'):
        # Create a new window to display the receipt
        print_window = tk.Toplevel(self.root)
        print_window.title("Receipt")
        print_window.geometry("350x300")

        # Set padding and font for the receipt
        padding = 10
        receipt_font = ("Helvetica", 12)

        # Create a frame for styling
        frame = tk.Frame(print_window, padx=padding, pady=padding)
        frame.pack(padx=padding, pady=padding)

        # Display receipt details with styling
        tk.Label(frame, text="----- Receipt -----",
font=("Helvetica", 14, "bold")).grid(row=0, column=0, columnspan=2,
pady=10)
```



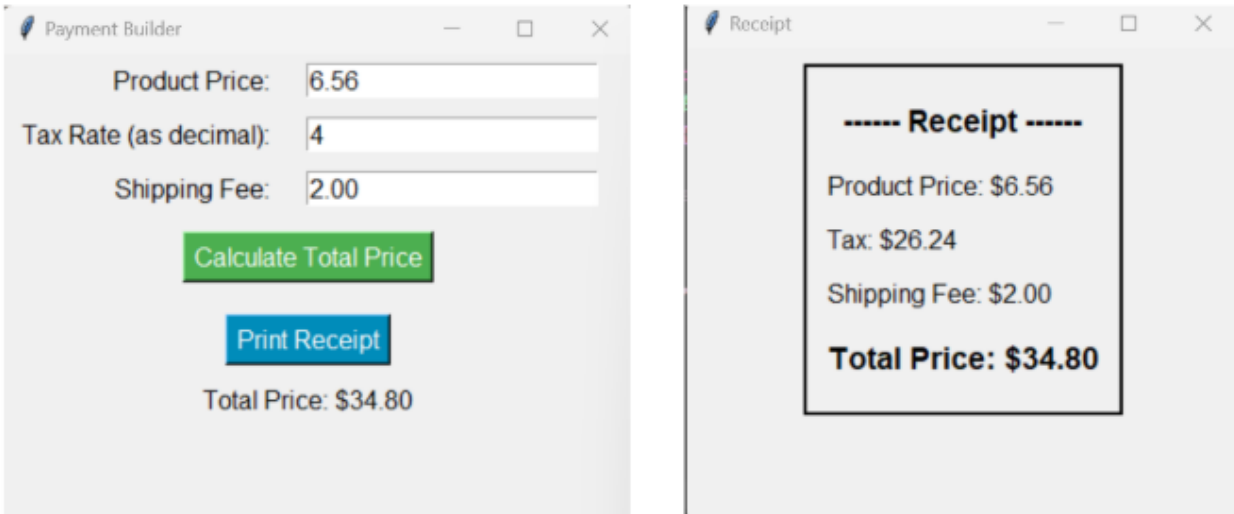
```
        tk.Label(frame, text=f"Product Price:
${self.receipt.productPrice():.2f}", font=receipt_font).grid(row=1,
column=0, sticky="w", pady=5)
        tk.Label(frame, text=f"Tax: ${self.receipt.addedTax():.2f}",
font=receipt_font).grid(row=2, column=0, sticky="w", pady=5)
        tk.Label(frame, text=f"Shipping Fee:
${self.receipt.shippingCost():.2f}", font=receipt_font).grid(row=3,
column=0, sticky="w", pady=5)
        tk.Label(frame, text=f"Total Price:
${self.receipt.totalPrice():.2f}", font=("Helvetica", 14,
"bold")).grid(row=4, column=0, columnspan=2, pady=10)

        # Add a border to the window for better visual appeal
        frame.config(bd=2, relief="solid")
    else:
        messagebox.showerror("No Calculation", "Please calculate the
total price first.")

# Main function to run the GUI
if __name__ == "__main__":
    root = tk.Tk()
    app = PaymentBuilderGUI(root)
    root.mainloop()
```

Output

Based on figure 10.2 below, it shows the receipt confirmation page. It generates the details like product price, taxes, shipping fee as well as final total price.



Payment Builder

Product Price: 6.56

Tax Rate (as decimal): 4

Shipping Fee: 2.00

Calculate Total Price

Print Receipt

Total Price: \$34.80

Receipt

----- Receipt -----

Product Price: \$6.56

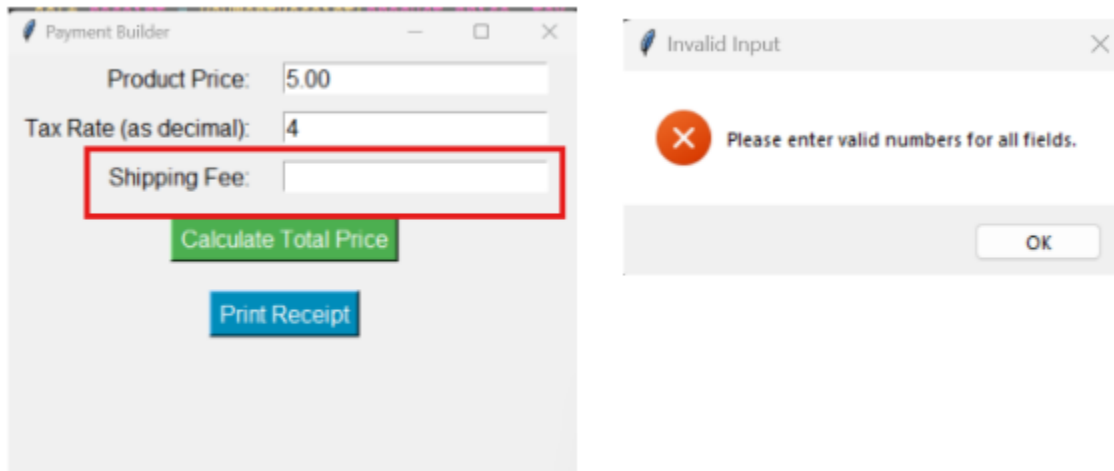
Tax: \$26.24

Shipping Fee: \$2.00

Total Price: \$34.80

Figure 10.2: Receipt Confirmation Page

The error message will appear when there is an empty field. It shows the “Invalid Input” page which notifies the user to enter the valid numbers for all fields as shown in figure 10.3 below.



Payment Builder

Product Price: 5.00

Tax Rate (as decimal): 4

Shipping Fee:

Calculate Total Price

Print Receipt

Invalid Input

Please enter valid numbers for all fields.

OK

Figure 10.3: Error Page

5 Conclusion and Suggestions

5.1 Conclusion

In conclusion, our proposed Grocery E-Commerce platform is designed with ease of use for the customer and the retailer in mind. By following the design principles and design patterns proposed, it will help us develop the software platform that can satisfy the requirements of both the customers and retailer, while providing a distinctive advantage that allows it to stand out from most e-commerce platforms. The platform is intended for a singular grocery store's use, to allow seamless integration of their delivery services, providing convenience for the customer. However, we envision this model to be applicable for various grocery stores.

5.2 Suggestions

A proposed concept that can provide service benefits for the customer, is Grocery Templates. Grocery Templates provides templates for customers to select, which will provide a grocery list of the type of items they'd need, and the customer can follow the guidelines to place their order. These templates can be based on various needs of the customer, such as family dinners, snack time, and other occasions, and can be tailored to different specific needs.

6 Bibliography/Reference

5.1 Bibliography

What is a Workflow Diagram? (n.d.). Lucidchart.

[What is a Workflow Diagram | Lucidchart](#)

Refactoring.Guru. (n.d.). *Design patterns*.

<https://refactoring.guru/design-patterns>

GeeksforGeeks. (2024b, October 16). *Observer design pattern*. GeeksforGeeks.

<https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/>

GeeksforGeeks. (2024, February 9). *Strategy design pattern*. GeeksforGeeks.

<https://www.geeksforgeeks.org/strategy-pattern-set-1/>

5.2 Reference

Santos, M., Gonçalves, P., & Oliveira, R. (2023). E-commerce: Issues, opportunities, challenges, and trends. Retrieved from

https://comum.rcaap.pt/bitstream/10400.26/50458/1/E-Commerce_-_Issues-Opportunities-Challenges-and-Trends.pdf

Le, S., Nguyen, T., & Vu, P. (2023). Extending the experience construct: An examination of online grocery shopping. *European Journal of Marketing*, 57(3), 365-390. Retrieved from

[\(PDF\) Extending the experience construct: an examination of online grocery shopping](#)

Cheng, L., Zhou, W., & Li, X. (2020). Building customer trust in online grocery stores. *Journal of Research in Interactive Marketing*, 14(4), 543-559. Retrieved from

<https://www.emerald.com/insight/content/doi/10.1108/JRIM-02-2020-0035/full/html>